МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение высшего образования
«МОСКОВСКИЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ (МАДИ)»
ВОЛЖСКИЙ ФИЛИАЛ МАДИ

УТВЕРЖДАЮ
Зав. кафедрой ИиТТП
Петрова А.В.

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ ПО ДИСЦИПЛИНЕ «ОБЛАЧНЫЕ ВЫЧИСЛЕНИЯ, ОБЛАЧНЫЕ ПРОГРАММНЫЕ ПРИЛОЖЕНИЯ И СЕРВИСЫ»

Направление подготовки
09.03.01 «Информатика и вычислительная техника»

Направленность (профиль)
Автоматизированные системы обработки информации и управления в отраслях транспортно-дорожного комплекса

Квалификация
Бакалавр

Кафедра: ИиТТП

Чебоксары

Содержание

ВВЕДЕНИЕ

Технологии **больших данных** – это совокупность методов и инструментов обработки данных больших объемов, различной структуры, получаемых от разнородных источников с большой скоростью. Чёткого определения больших данных не придумано, но используются характеристики задачи обработки больших данных, обозначаемые обычно 3V (4V, 5V, 7V), по первым буквам англоязычных терминов:

- Volume – данные имею большой объем,
- Velocity – данные поступают с большой скоростью и требуют скоростной обработки, в реальном или полу-реальном времени,
- Variety – данные имеют разнообразную структуру и могут быть неструктурированными или полу-структурированными,
- Veracity – данные имеют различную достоверность,
- Viability – жизнеспособность данных,
- Value – из данных, в результате обработки, необходимо извлечь некую ценность для бизнеса,
- Variability как переменчивость данных,
- Visualisation как необходимость представления результатов обработки данных в человеко-читаемом виде, визуализации данных.

Одним из популярных фреймворков для обработки больших данных является Apache Hadoop, включающий в себя набор подсистем для распределенного хранения, обработки. импорта/экспорта данных.

Предлагаемые лабораторные работы дают возможность получить практические навыки использования Apache Hadoop и Apache Spark для решения задач обработки больших данных, а также, в лабораторной работе №6, познакомиться с основами контейнеризации приложений с помощью системы Docker.

Для выполнения лабораторных работ №№1 – 3 используется виртуальная машина Cloudera Quickstart VM, которая содержит предварительно развёрнутый и настроенный кластер Apache Hadoop. Работы №№ 4 и 5 посвящены изучению Apache Spark, для их выполнения виртуальная машина Cloudera Quickstart VM не требуется.

По итогам выполнения каждой лабораторной работы оформляется отчет, включающий следующие разделы:

1. Цель работы;
2. Задание на работу;

3. Исходный код программы;
4. Результаты выполнения программы;
5. Выводы по результатам работы.

Оформленный отчет представляется к защите, которая проходит в форме устного опроса по материалам лабораторной работы.

ОПИСАНИЕ ЛАБОРАТОРНОЙ УСТАНОВКИ

Лабораторная установка состоит из персонального компьютера, на котором установлена и настроена виртуальная машина Cloudera Quickstart VM под управлением Virtual Box. Cloudera Quickstart VM содержит установленный и готовый к работе кластер Apache Hadoop, включающий один узел. Использование Cloudera Quickstart VM избавляет от необходимости вручную разворачивать и настраивать кластер Apache Hadoop.

Ниже приведен порядок установки и настройки системы:

1. При отсутствии на компьютере Virtual Box скачать его с сайта https://www.virtualbox.org/ и установить.

2. Скачать виртуальную машину Cloudera Quickstart VM с сайта https://www.cloudera.com/downloads/quickstart_vms/5-12.html (выбрать платформу Virtual Box).

3. Открыть Virtual Box и создать виртуальную машину с параметрами, указанными на рисунках 1 и 2, размером памяти не менее 4096 Мб.

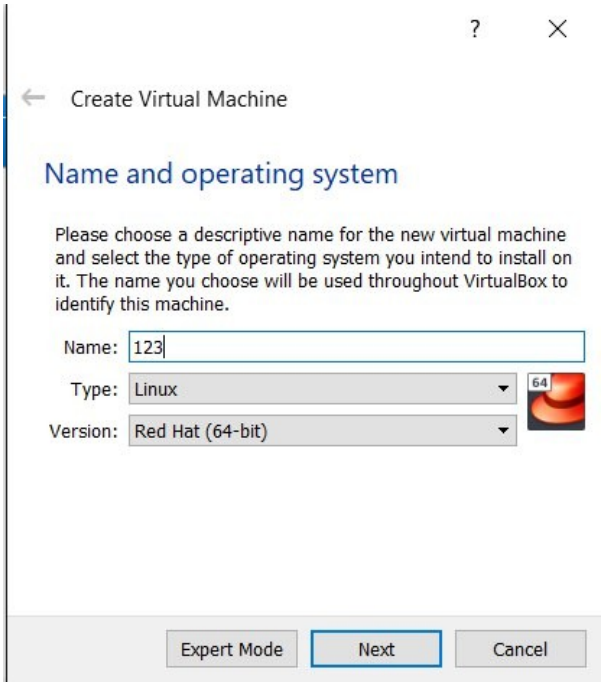


Рисунок 1 – Параметры виртуальной машины

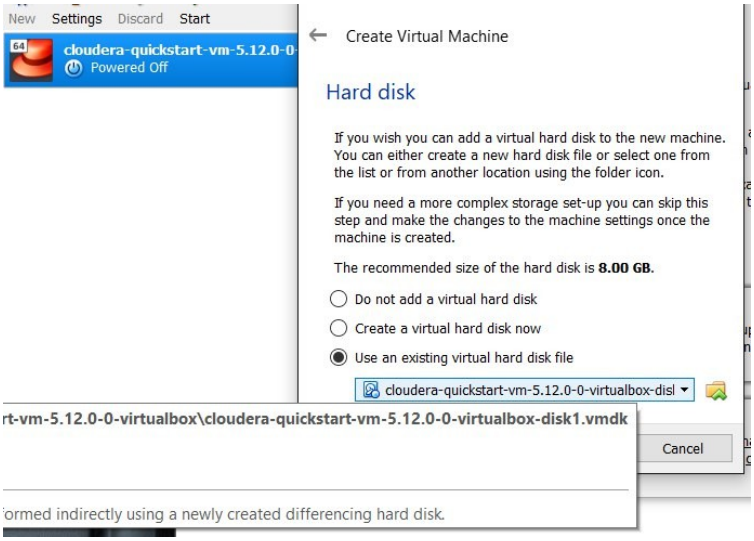


Рисунок 2 – Параметры виртуальной машины

Cloudera необходимо разместить в папке, предлагаемой по умолчанию, предварительно распаковав. После старта виртуальной машины лабораторная установка готова к использоваию.

1. ЛАБОРАТОРНАЯ РАБОТА №1
ИССЛЕДОВАНИЕ СПОСОБОВ ИСПОЛЬЗОВАНИЯ ЭКОСИСТЕМЫ APACHE HADOOP

1. ЦЕЛЬ РАБОТЫ

Исследовать возможности основных компонентов экосистемы Apache Hadoop. Получить практические навыки использования экосистемы Apache Hadoop для выборки структурированных данных.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

Hadoop – это проект с открытым исходным кодом, находящийся под управлением Apache Software Foundation (http://hadoop.apache.org/). Hadoop используется для надежных, масштабируемых и распределенных вычислений, но может также применяться и как хранилище файлов общего назначения, способное вместить петабайты данных. Многие компании используют Hadoop в исследовательских и производственных целях.

Hadoop состоит из двух ключевых компонентов:

- распределенная файловая система Hadoop (англ. HDFS, Hadoop Distributed File System), которая отвечает за хранение данных на кластере Hadoop;
- система MapReduce, предназначенная для вычислений и обработки больших объемов данных на кластере.

В состав Hadoop также входят следующие компоненты:

- Hadoop Common – библиотеки и утилиты;
- Hadoop YARN – платформа управления ресурсами и планирования.

На основе этих ключевых компонентов создано несколько подпроектов, таких как:

- HBase – масштабируемое хранилище данных с поддержкой объемных таблиц;
- Hive – инфраструктура хранилища данных, которая обеспечивает сводку данных и специальные запросы;
- Pig – высокоуровневый язык потока данных и фреймворк для

параллельных вычислений;

- Spark – быстрый вычислительный механизм для обработки данных экосистемы Hadoop.

2.1. Развитие Hadoop

На рисунке 1 показаны основные отличия архитектуры Hadoop версии 1.0 и версии 2.0. В Hadoop 2.0 из подсистемы MapReduce выделена подсистема YARN, которая выполняет задачи по управлению ресурсами кластера Hadoop, и подсистема Tez, реализующая набор функций для выполнения вычислений над большими данными, хранящимися в HDFS. Кроме того, изменения претерпела распределённая файловая система HDFS, о чем подробней будет сказано в лабораторной работе №2.

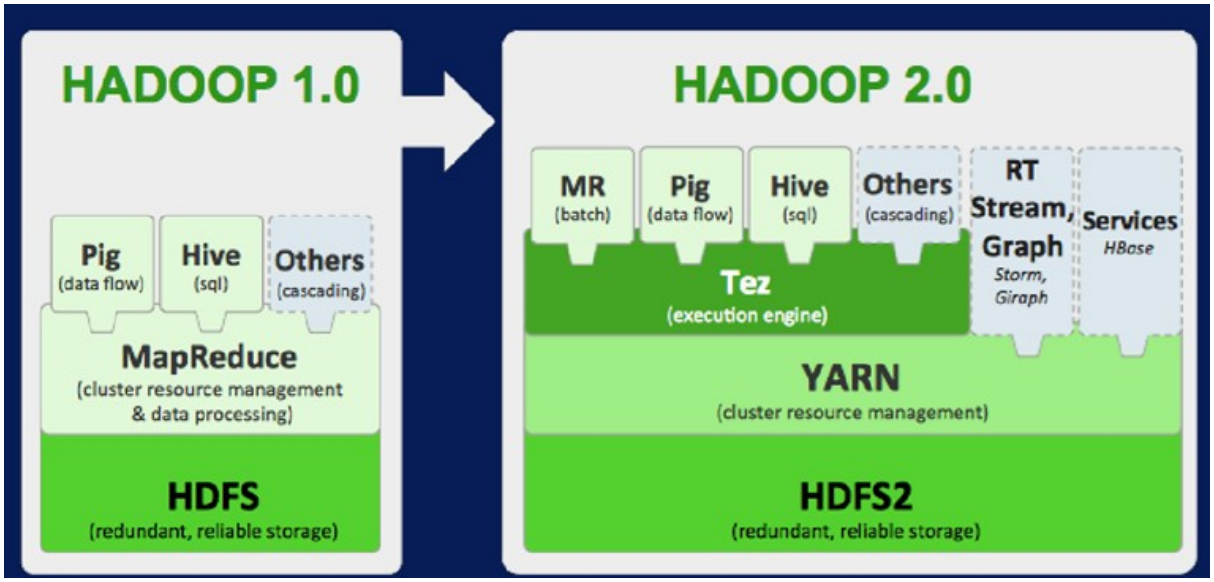


Рисунок 1 – Компоненты Hadoop 1.0 и 2.0

2.2. Подсистема Apache Pig

Apache Pig представляет собой абстракцию над MapReduce. Pig – это платформа, которая используется для анализа больших наборов данных, представленных как потоки данных. Для написания программ (скриптов) анализа, чтения, записи, обработки данных Pig предоставляет SQL-подобный язык высокого уровня Pig Latin. Для выполнения эти программы преобразуются в Map и Reduce задачи с помощью модуля Pig Engine. К

достоинствам Pig Latin можно отнести:

1. **Богатый набор операторов.** Он предоставляет множество операторов объединения, сортировки, фильтрации данных и т. д.

2. **Простота программирования.** Pig Latin похож на SQL, что позволяет легко написать скрипт для Pig, если вы хорошо разбираетесь в SQL.

3. Возможности оптимизации. Задачи в Apache Pigавтоматически оптимизируются при выполнении, поэтому программистам необходимо сосредоточиться только на семантике языка.

4. Расширяемость. Используя существующие операторы, пользователи могут создавать свои собственные функции для чтения, обработки и записи данных.

5. UDF's. Pig предоставляет возможность создавать пользовательские функции на других языках программирования, таких как Java, и вызывать или встраивать их в скрипты Pig.

6. Обрабатывает все виды данных. Apache Pig позволяет анализировать данных различного вида, как структурированные, так и неструктурированные. Он сохраняет результаты в HDFS.

По сравнению с MapReduce Pig Latin предоставляет более высокоуровневые возможности. По сравнению с SQL Pig Latin является процедурным, а не декларативным языком, позволяет обрабатывать данные без разработки схемы данных, но имеет меньше возможностей для оптимизации запросов.

Модель данных Pig Latin схематично показана на рисунке 2.

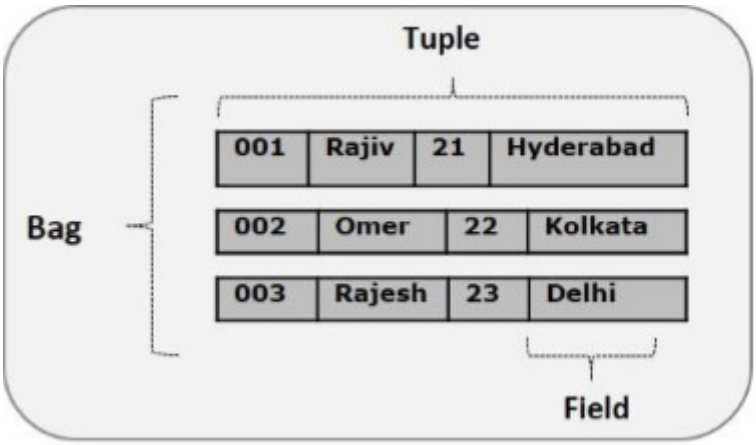


Рисунок 2 – Схематическое представление модели данных Pig Latin

В состав структуры данных входит **атом (field)** – любое единичное значение в языке Pig Latin, независимо от типа данных. Оно сохраняется как строка и

может использоваться как строка и номер. Атомарными значениями являются: int, long, float, double, chararray и bytearray. Атомарное значение называется **полем (field)**.

Кроме элементарных типов Pig поддерживает три составных типа данных: ассоциативные массивы (map), кортежи (tuple) и таблицы (bag).

Запись, образованная упорядоченным набором полей, называется **кортежем (tuple)**, поля могут быть любого типа. Кортеж аналогичен строке таблицы реляционной базы данных, например: (Иванов, 30).

Неупорядоченный набор кортежей, не обязательно уникальных, образует таблицу **(bag, «мешок»).** При этом каждый кортеж может иметь произвольное количество полей (гибкая схема), кортежи могут быть вложенными. Пример:

{(Иванов, 30), (Петров, 35, {2024, petrov@gmail.com})}.

Ассоциативный массив (map) представляет собой набор пар ключ-значение. Ключ должен быть типа chararray и должен быть уникальным, значение может быть любого типа. Пример: [name#Иванов, age#30].

Отношение (Relation) – это множество кортежей. Отношения в Pig Latin неупорядочены, не гарантируется порядок обработки кортежей в отношении.

Архитектура Apache Pig представлена на рисунке 3. К основным компонентам Apache Pig относятся:

1. Синтаксический анализатор (Parser). Первоначально сценарии Pig обрабатываются в модуле Parser. Он проверяет синтаксис скрипта, проверяет типизацию и выполняет другие проверки. Результатом работы анализатора будет DAG (directed acyclic graph) – ориентированный ациклический граф, который представляет собой инструкции Pig Latin и логические операторы. В DAG логические операторы скрипта представлены в виде узлов, а потоки данных представлены как дуги.

2. Оптимизатор (Optimizer). Логический план запроса в виде DAG передается логическому оптимизатору, который выполняет логическую оптимизацию.

3. Compiler. Компилятор компилирует оптимизированный логический план в серию задач MapReduce.

4. Execution engine. Наконец, задания MapReduce передаются в Hadoop в отсортированном порядке и выполняются, что дает желаемые результаты.

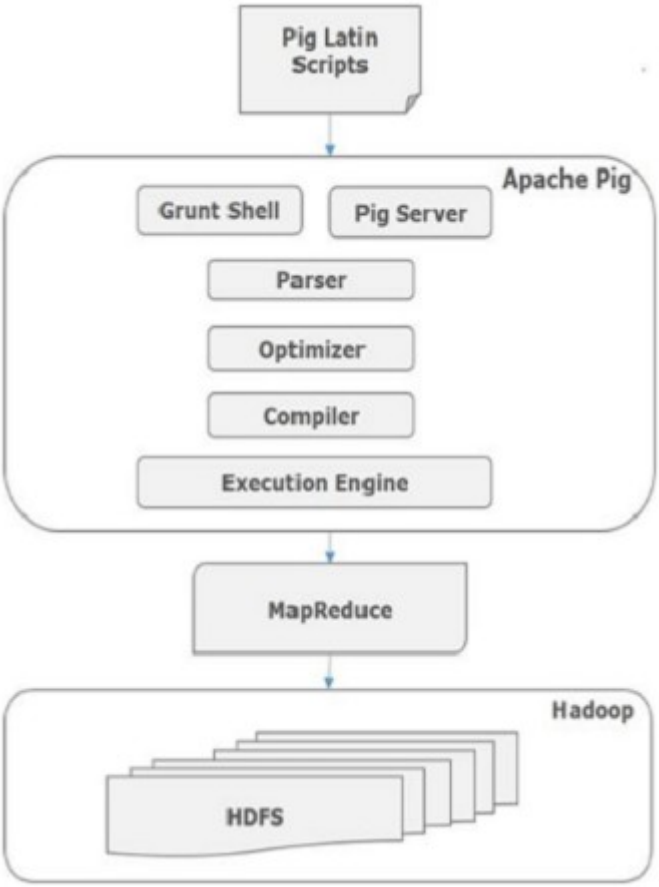


Рисунок 3 – Архитектура Apache Pig

2.3. Подсистема Apache Hive

Apache Hive – это СУБД, построенная поверх Apache Hadoop для предоставления обобщения, запроса и анализа данных. Hive предоставляет SQL-подобный интерфейс для запроса данных, хранящихся в различных базах данных и файловых системах, которые интегрируются с Hadoop. Традиционные SQL-запросы должны быть реализованы в API Java MapReduce для выполнения приложений и запросов SQL по распределенным данным. Hive предоставляет необходимую абстракцию SQL для интеграции SQL-подобных запросов (HiveQL) в базовую Java без необходимости реализовывать запросы в низкоуровневом Java API. Поскольку большинство приложений для хранилищ данных работают с SQL-ориентированными языками запросов, Hive помогает переносить приложения на основе SQL в Hadoop.

Архитектура Apache Hive приведена на рисунке 4.

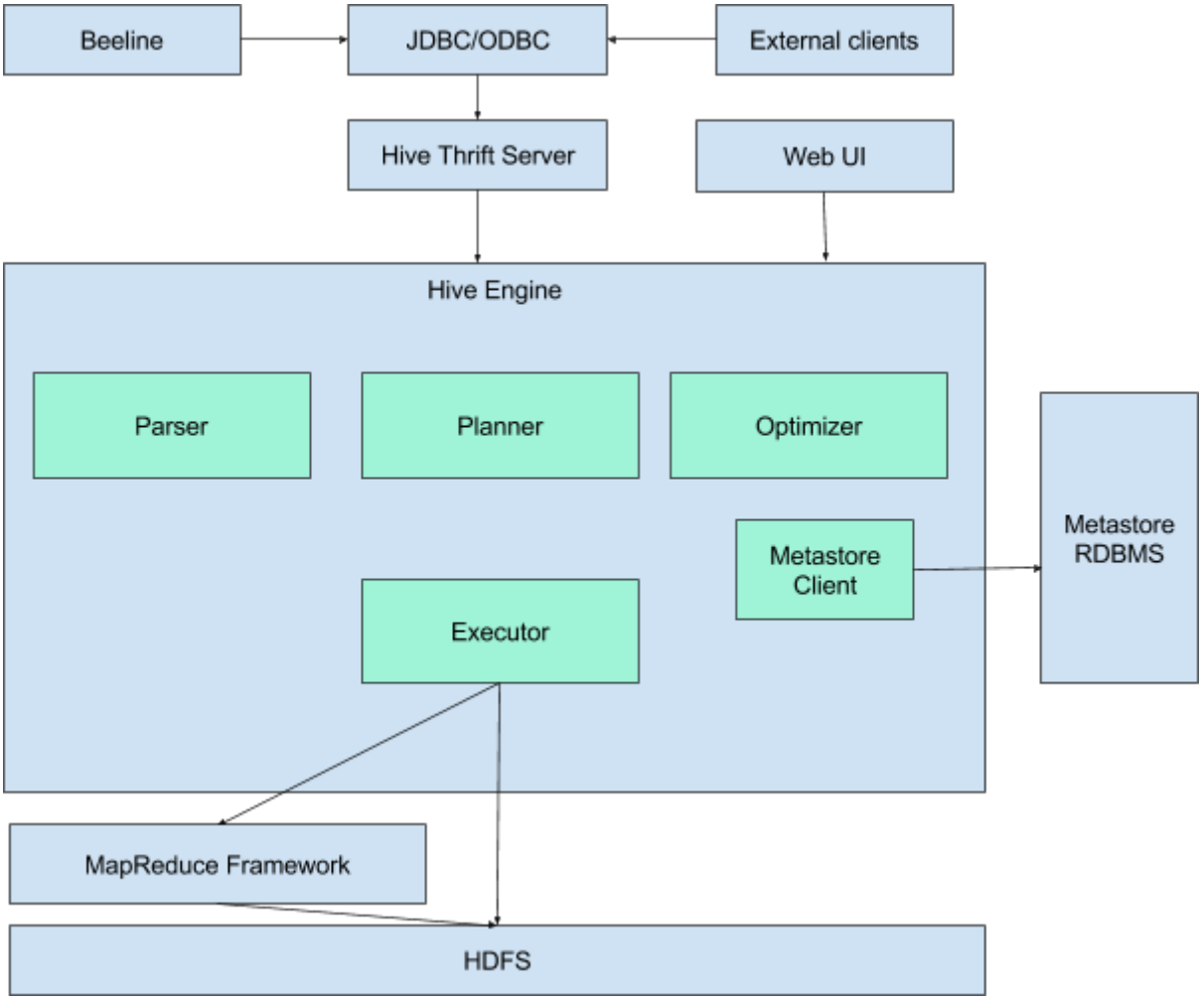


Рисунок 4 — Архитектура Apache Hive

Основными компонентами архитектуры Hive являются:

1. **Метахранилище**: для каждой таблицы сохраняет метаданные, такие как схема и местоположение. Также включает метаданные разделов, которые помогают драйверам отслеживать прогресс различных наборов данных, распределенных по кластеру. Данные хранятся в традиционном формате реляционных баз данных. Метаданные помогают драйверу отслеживать данные, что очень важно, т.к. резервный сервер регулярно реплицирует данные, которые могут быть извлечены в случае потери данных.

2. **Драйвер**: действует как контроллер, который получает инструкции HiveQL. Он запускает выполнение инструкции, создавая сеансы и контролируя жизненный цикл и ход выполнения запроса. Он хранит необходимые метаданные, созданные во время выполнения инструкции HiveQL. Драйвер также действует как точка сбора данных или результатов запроса.

3. **Компилятор**: выполняет компиляцию запроса HiveQL, преобразует запрос в план выполнения. Этот план содержит задачи и шаги, которые

должен выполнить Hadoop MapReduce, чтобы получить результат. Компилятор преобразует запрос в абстрактное синтаксическое дерево (AST). После проверки совместимости и ошибок времени компиляции он преобразует AST в ориентированный ациклический граф (DAG). DAG делит операторы на этапы и задачи MapReduce на основе входных запросов и данных.

4. **Оптимизатор**: выполняет различные преобразования в плане выполнения запроса, чтобы получить оптимизированный DAG. Преобразования могут быть объединены вместе, например, преобразование конвейера соединений в одно соединение для лучшей производительности. Он также может разбивать задачи, чтобы обеспечить лучшую производительность и масштабируемость.

5. **Исполнитель**: после компиляции и оптимизации исполнитель выполняет задачи. Он взаимодействует с диспетчером задач Hadoop для планирования задач, которые должны выполняться. Он организует конвейерную обработку задач, следя за тем, чтобы задача с зависимостями выполнялась только при выполнении всех предварительных условий.

6. **CLI, UI и Thrift Server**: интерфейс командной строки (CLI) предоставляет пользовательский интерфейс для внешнего пользователя для взаимодействия с Hive путем отправки запросов, инструкций и для мониторинга состояния процесса. Сервер Thrift позволяет внешним клиентам взаимодействовать с Hive по сети, аналогично протоколам JDBC или ODBC.

3. МЕТОДИКА ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

3.1. Установить и запустить виртуальную машину Cloudera Quickstart VM.

3.2. В браузере открыть стартовую страницу по адресу quickstart.cloudera, как показано на рисунке 5.

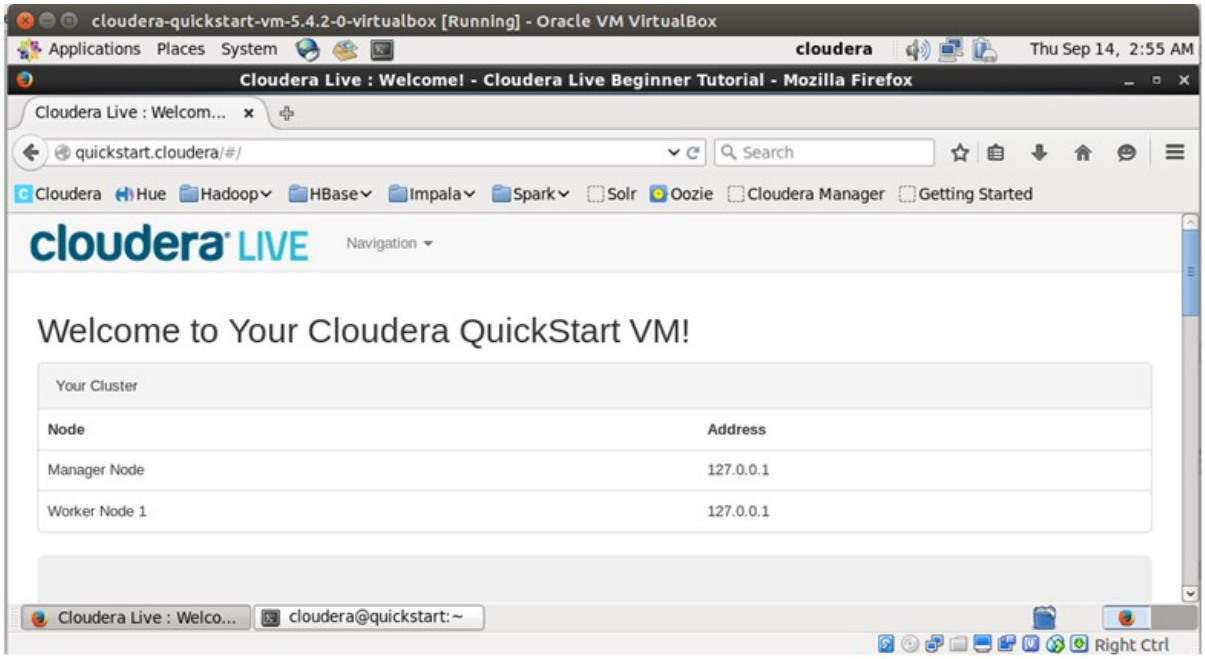


Рисунок 5 – Стартовая страница

3.3. Ознакомиться с доступными разделами Cloudera Quickstart VM.

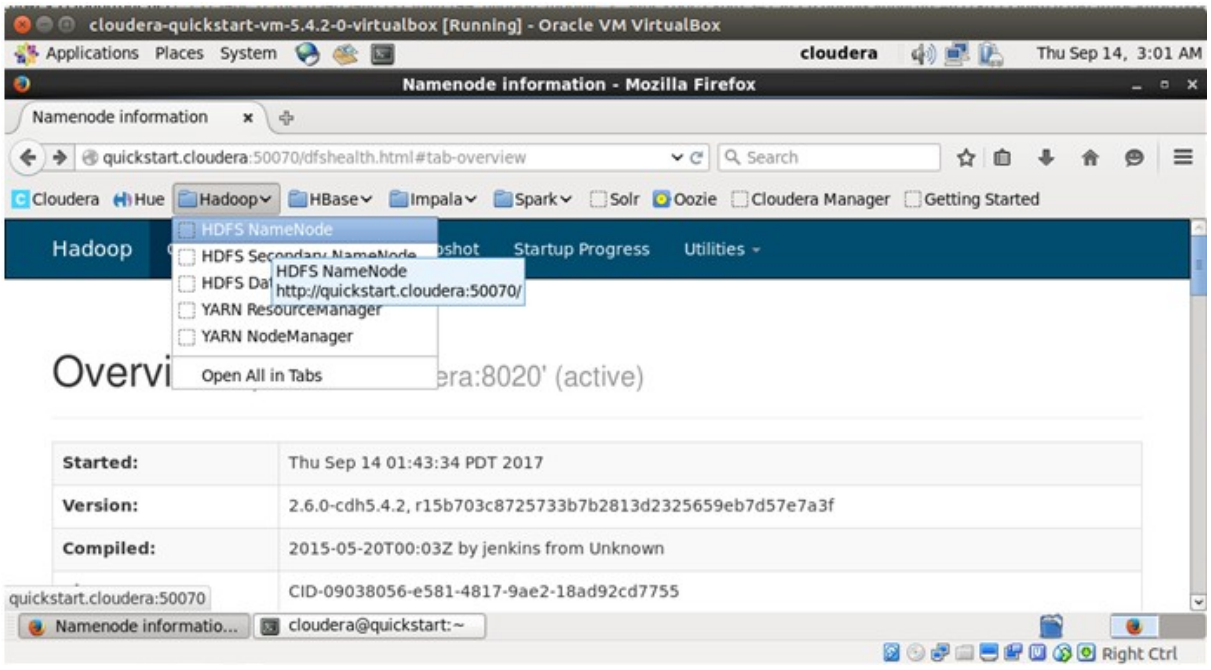


Рисунок 6 – Основная информация о Hadoop HDFS NameNode

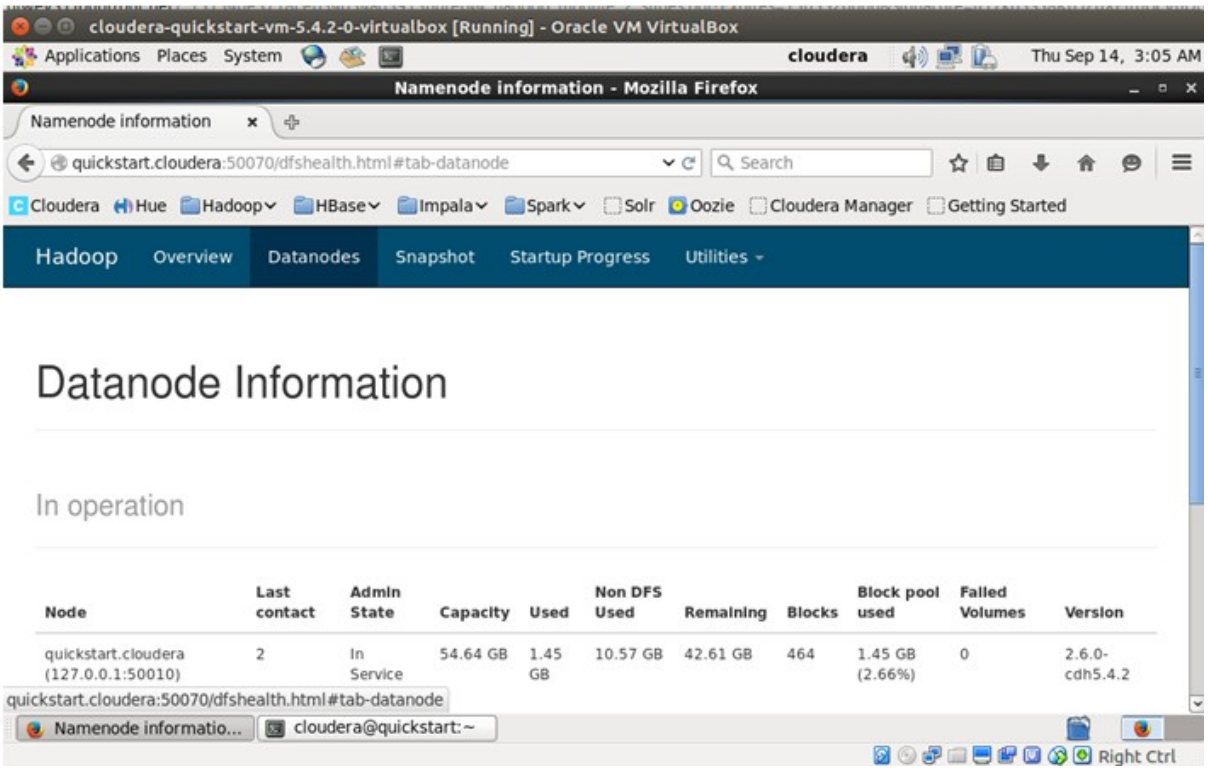


Рисунок 7 – Информация о DataNode

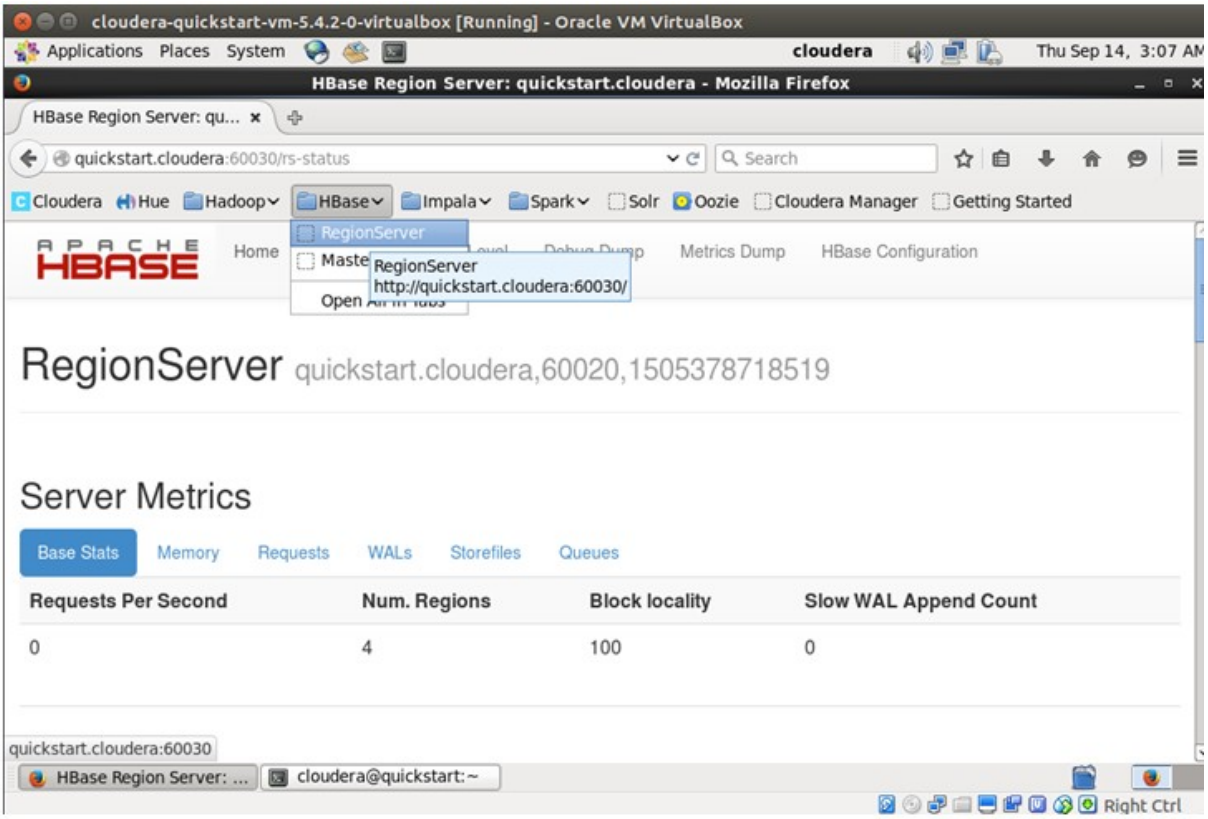


Рисунок 8 – HBase

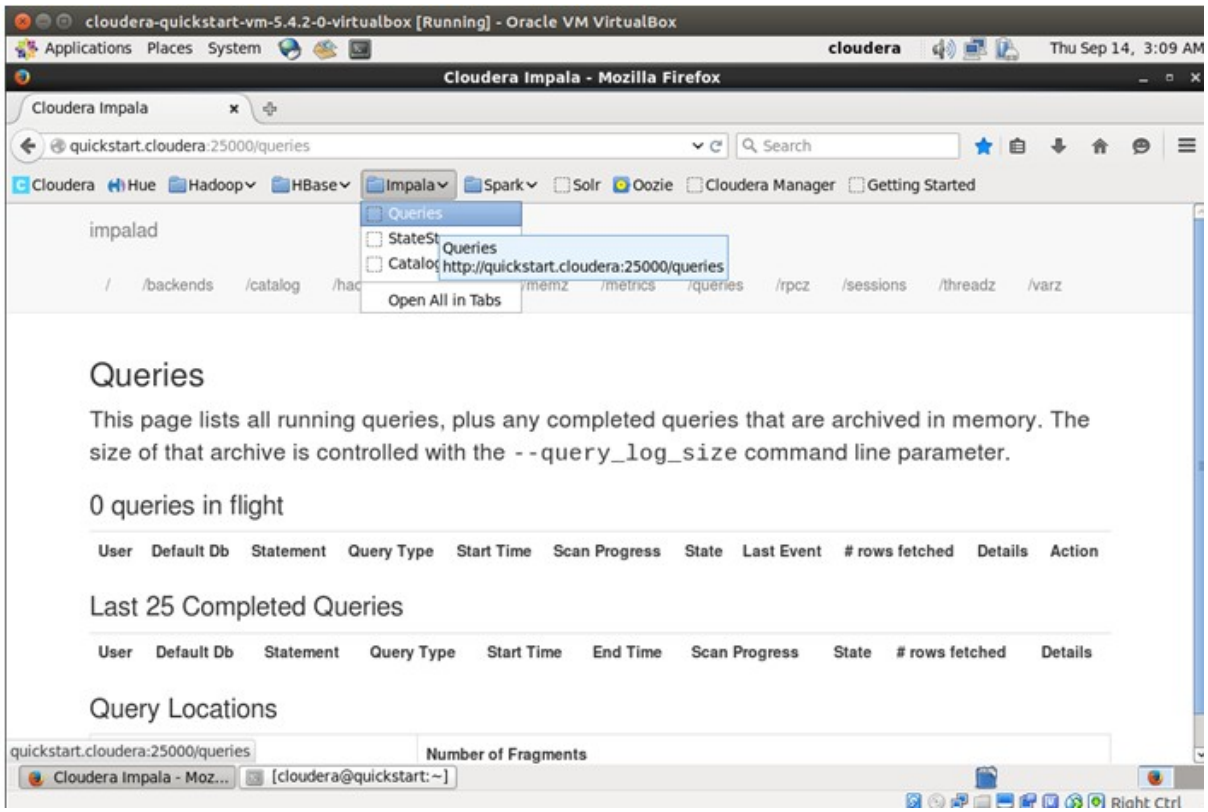


Рисунок 9 – Impala

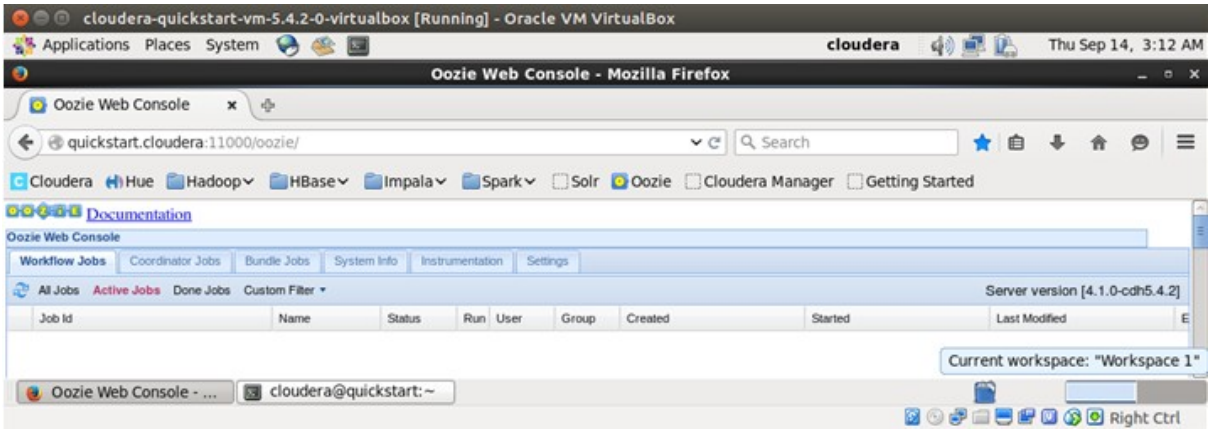


Рисунок 10 – Oozie

3.4. Экспортировать данные из СУБД в HDFS с помощью утилиты Apache Sqoop. Sqoop позволяет экспортировать данные из БД MySQL, сохранив их структуру.

```
sqoop import-all-tables \
-m 1 \
--connect jdbc:mysql://quickstart:3306/retail_db \
--username=retail_dba \
--password=cloudera \
--compression-codec=snapy \
```

--as-avrodatafile \
--warehouse-dir=/user/hive/warehouse
В результате будут запущены задачи MapReduce для экспорта данных MySQL в файлы в формате Apache Avro. Файлы с данными и схема данных будут помещены в HDFS.

После завершения экспорта можно просмотреть результат с помощью следующих команд:
hadoop fs -ls /user/hive/warehouse
hadoop fs -ls /user/hive/warehouse/categories

```
[cloudera@quickstart ~]$ hadoop fs -ls /user/hive/warehouse
Found 6 items
drwxr-xr-x   - cloudera hive          0 2018-02-26 04:36 /user/hive/warehouse/categories
drwxr-xr-x   - cloudera hive          0 2018-02-26 04:37 /user/hive/warehouse/customers
drwxr-xr-x   - cloudera hive          0 2018-02-26 04:37 /user/hive/warehouse/departments
drwxr-xr-x   - cloudera hive          0 2018-02-26 04:38 /user/hive/warehouse/order_items
drwxr-xr-x   - cloudera hive          0 2018-02-26 04:38 /user/hive/warehouse/orders
drwxr-xr-x   - cloudera hive          0 2018-02-26 04:39 /user/hive/warehouse/products
[cloudera@quickstart ~]$ hadoop fs -ls /user/hive/warehouse/categories
Found 2 items
-rw-r--r--   1 cloudera hive          0 2018-02-26 04:36 /user/hive/warehouse/categories/_SUCCESS
-rw-r--r--   1 cloudera hive       1502 2018-02-26 04:36 /user/hive/warehouse/categories/part-m-00000.a
vro
```

Рисунок 11 – Результаты экспорта данных СУБД

При экспорте созданы также .avsc-файлы со схемами данных в домашнем каталоге:
ls -l *.avsc

```
[cloudera@quickstart ~]$ ls -1 *.avsc
sqoop_import_categories.avsc
sqoop_import_customers.avsc
sqoop_import_departments.avsc
sqoop_import_order_items.avsc
sqoop_import_orders.avsc
sqoop_import_products.avsc
```

Рисунок 12 – Схемы экспортированных данных

Пример схемы данных sqoop_import_categories.avsc:

```
{
  "type" : "record",
  "name" : "sqoop_import_categories",
  "doc" : "Sqoop import of categories",
  "fields" : [ {
    "name" : "category_id",
    "type" : [ "int", "null" ],
    "columnName" : "category_id",
```

```
  "sqlType" : "4"
 }, {
  "name" : "category_department_id",
  "type" : [ "int", "null" ],
  "columnName" : "category_department_id",
  "sqlType" : "4"
 }, {
  "name" : "category_name",
  "type" : [ "string", "null" ],
  "columnName" : "category_name",
  "sqlType" : "12"
 } ],
 "tableName" : "categories"
}
```


Схемы данных используются только при выполнении запросов (технология schema-on-read). То есть данные интерпретируются в момент выполнения запроса.

Для дальнейшего использования утилитой Apache Hive скопируем схемы данных в HDFS:

```
sudo -u hdfs hadoop fs -mkdir /user/examples
sudo -u hdfs hadoop fs -chmod +rw /user/examples
hadoop fs -copyFromLocal ~/*.avsc /user/examples
```

3.5. Выполнить выборку структурированных данных с помощью утилиты Impala. Для доступ к Impala следует использовать Hue, который предоставляет веб-интерфейс к ряду утилит Hadoop. Hue доступен по адресу 127.0.0.1:8888.

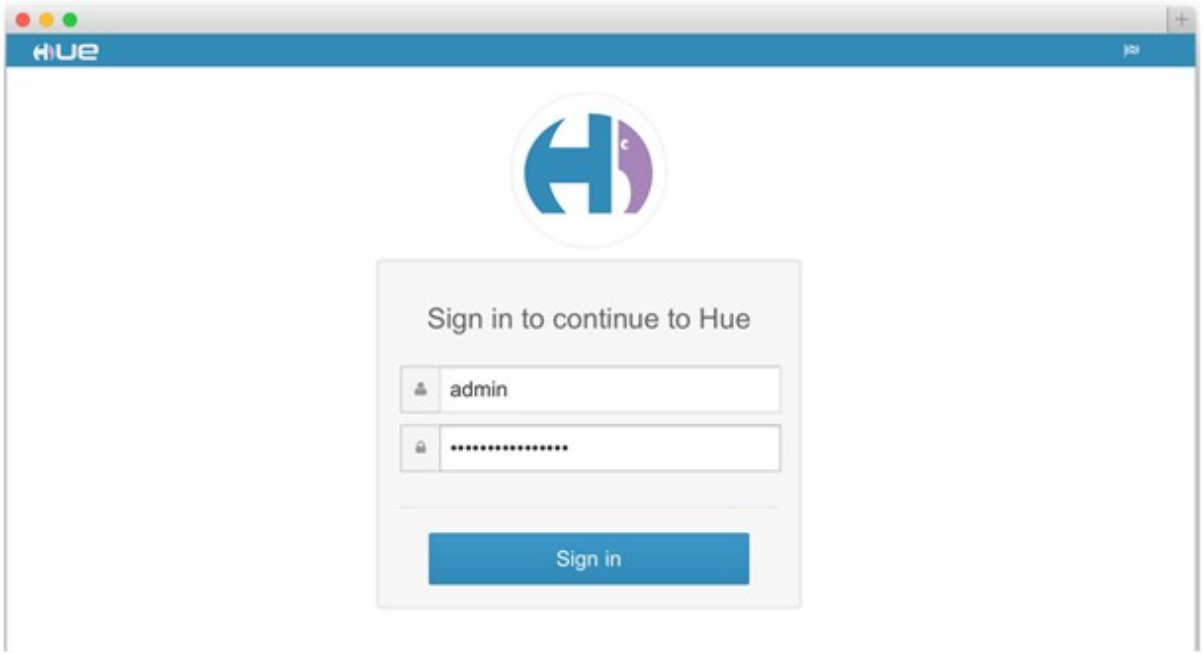


Рисунок 13 – Форма входа в Hue

После входа (рисунок 13, логин: cloudera/пароль: cloudera) со стартовой страницы Hue следует перейти в Query Editors/Impala, как показано на рисунке 14.

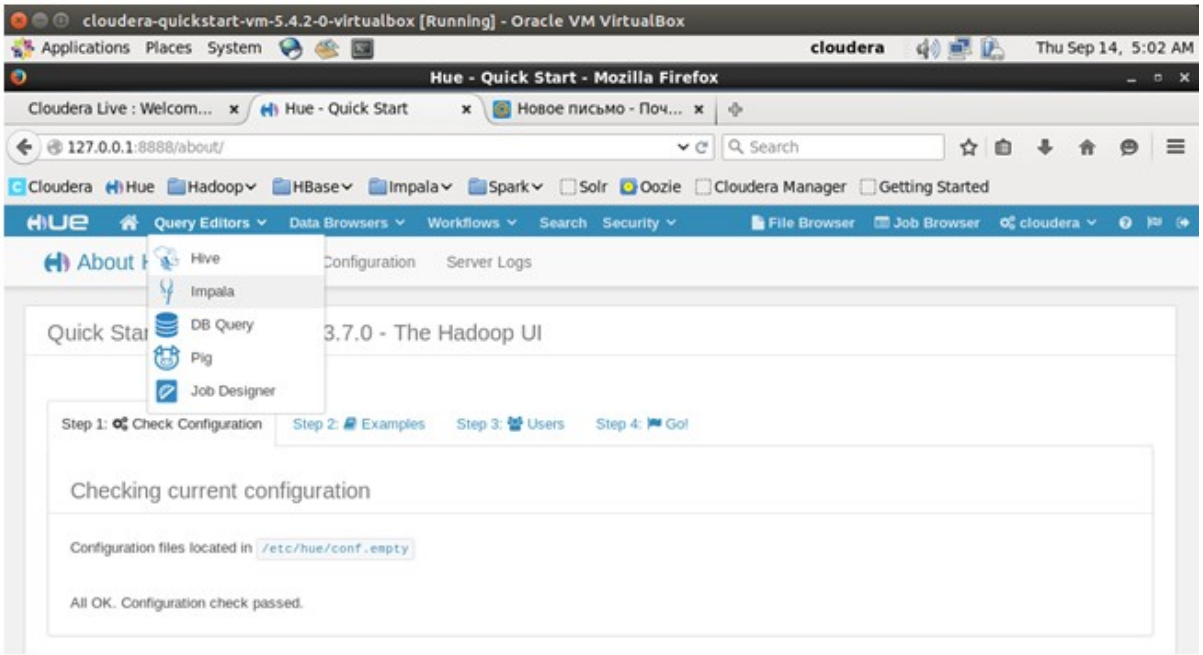


Рисунок 14 – Стартовая страница Hue

На странице Impala необходимо выполнить запросы для создания таблиц на основе ранее экспортированных данных:
CREATE EXTERNAL TABLE categories STORED AS AVRO

```
LOCATION 'hdfs:///user/hive/warehouse/categories'
TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_categories.avsc')
;

CREATE EXTERNAL TABLE customers STORED AS AVRO
LOCATION 'hdfs:///user/hive/warehouse/customers'
TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_customers.avsc')
;

CREATE EXTERNAL TABLE departments STORED AS AVRO
LOCATION 'hdfs:///user/hive/warehouse/departments'
TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_departments.avsc');

CREATE EXTERNAL TABLE orders STORED AS AVRO
LOCATION 'hdfs:///user/hive/warehouse/orders'
TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_orders.avsc');

CREATE EXTERNAL TABLE order_items STORED AS AVRO
LOCATION 'hdfs:///user/hive/warehouse/order_items'
TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_order_items.avsc');

CREATE EXTERNAL TABLE products STORED AS AVRO
LOCATION 'hdfs:///user/hive/warehouse/products'
                                TBLPROPERTIES
 ('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_products.avsc');
```

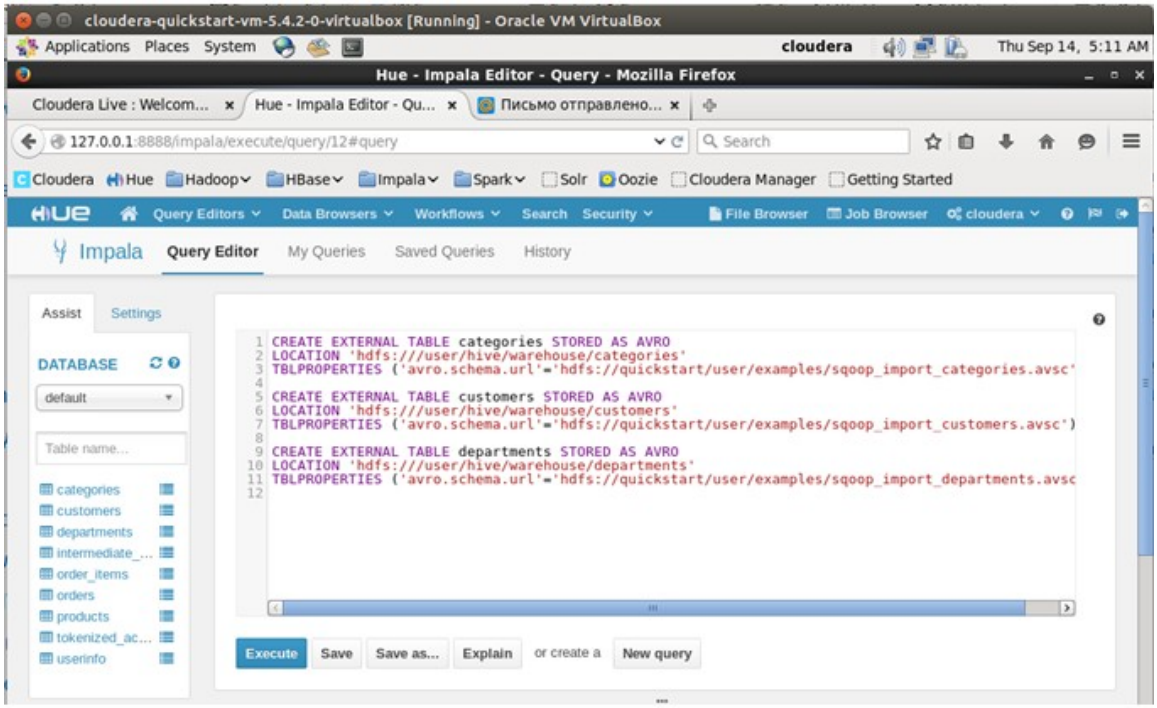


Рисунок 15 – Создание таблиц

В качестве примера выполним выборку 10 самых популярных категорий продуктов:

```
-- Most popular product categories
select c.category_name, count(order_item_quantity) as count
from order_items oi
inner join products p on oi.order_item_product_id = p.product_id
inner join categories c on c.category_id = p.product_category_id
group by c.category_name
order by count desc
limit 10;
```

3.6. Использовать возможности Pig для работы с данными.

Пример использования Pig

```
hdfs dfs -put  /etc/passwd  /user/cloudera
pig -x mapreduce
```

Эти команды поместят вас в оболочку Pig.

Загрузите файл: load '/ user / cloudera / passwd', используя PigStorage (':');

Выберите подмножество значений, как показано на рисунке 16.

```
grunt> A = load '/user/cloudera/passwd' using PigStorage(':');
grunt> B = foreach A generate $0, $4, $5 ;
grunt> dump B;
```

Рисунок 16 – Работа с Pig

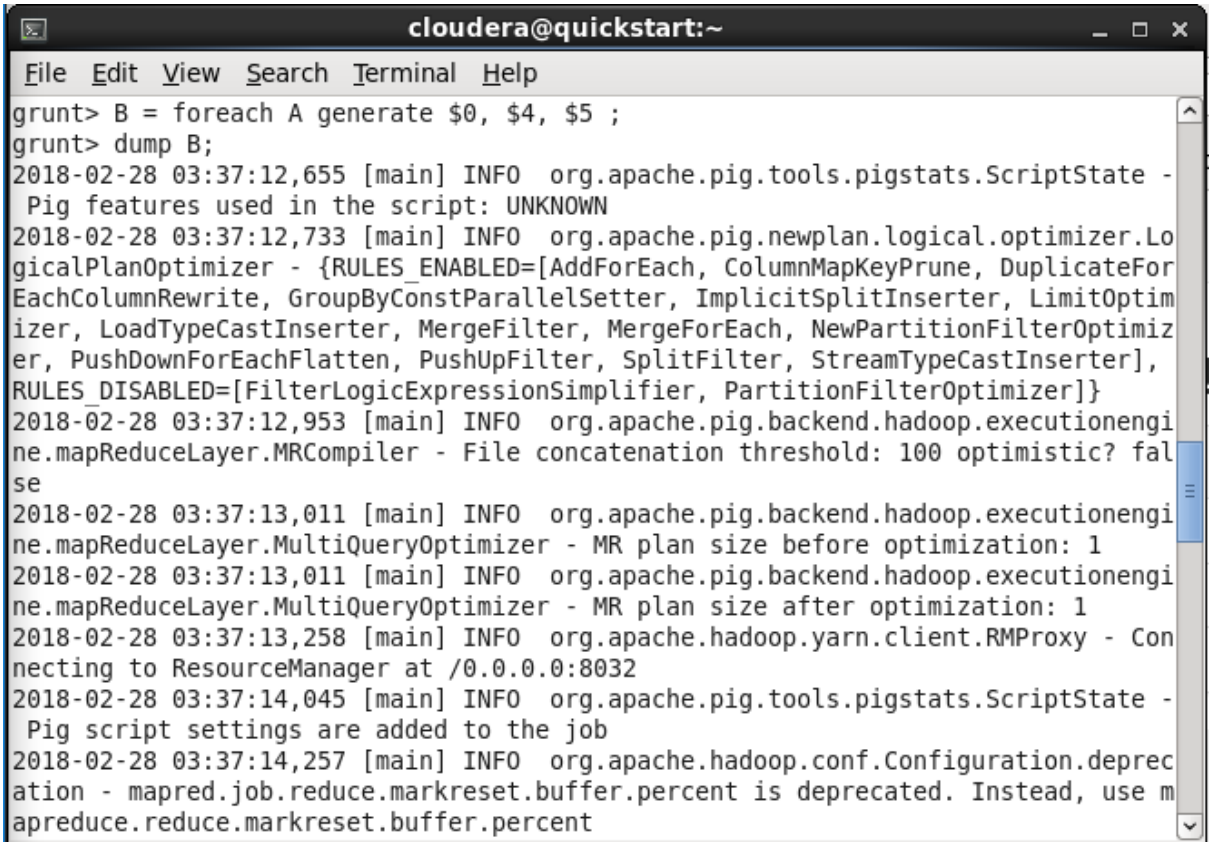


Рисунок 17 – Информация о работе Hadoop

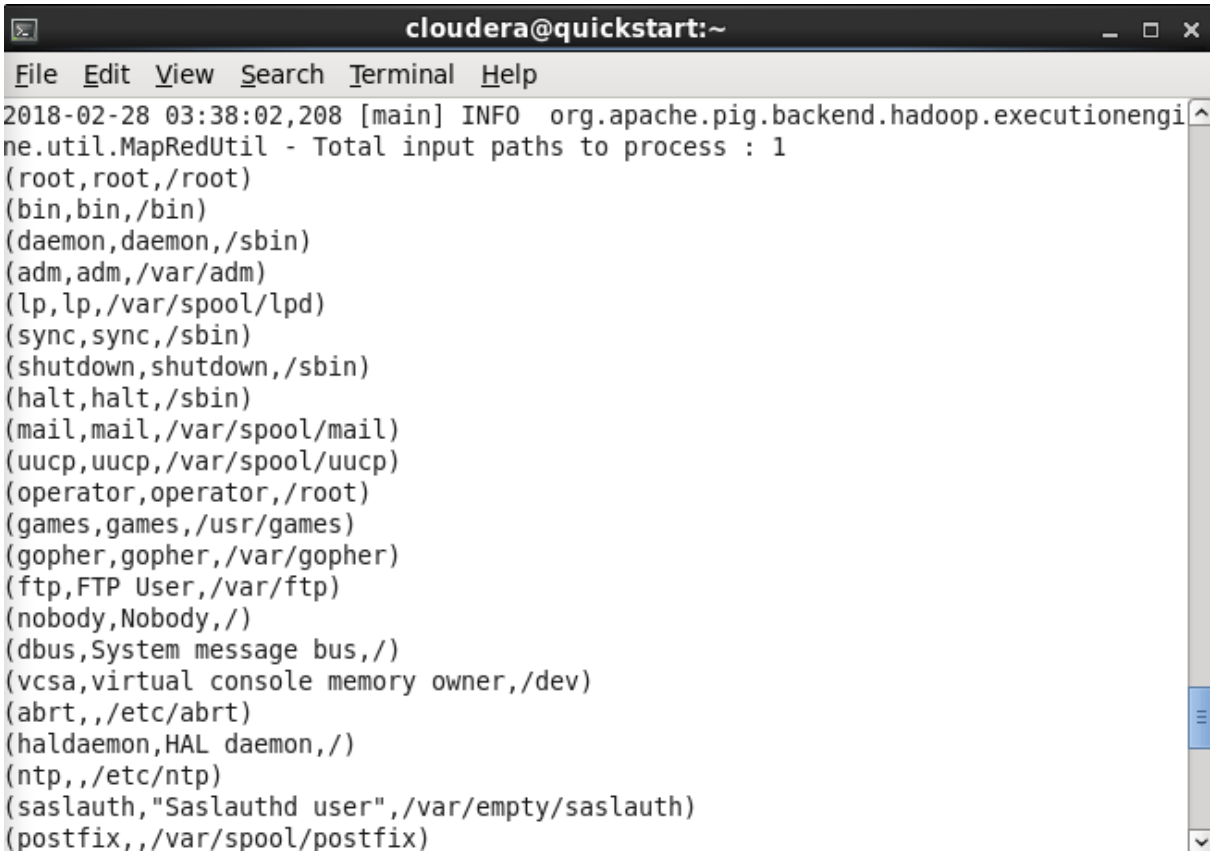


Рисунок 18 – Выводит имя пользователя, полное имя и домашний каталог

Можно сохранить эту информацию командой:
store B into ‘userinfo.out’;
Проверьте наличие новой информации в HDFS командой:
hadoop fs -ls /user/hive/warehouse
Предварительно покинув grunt командой quit.

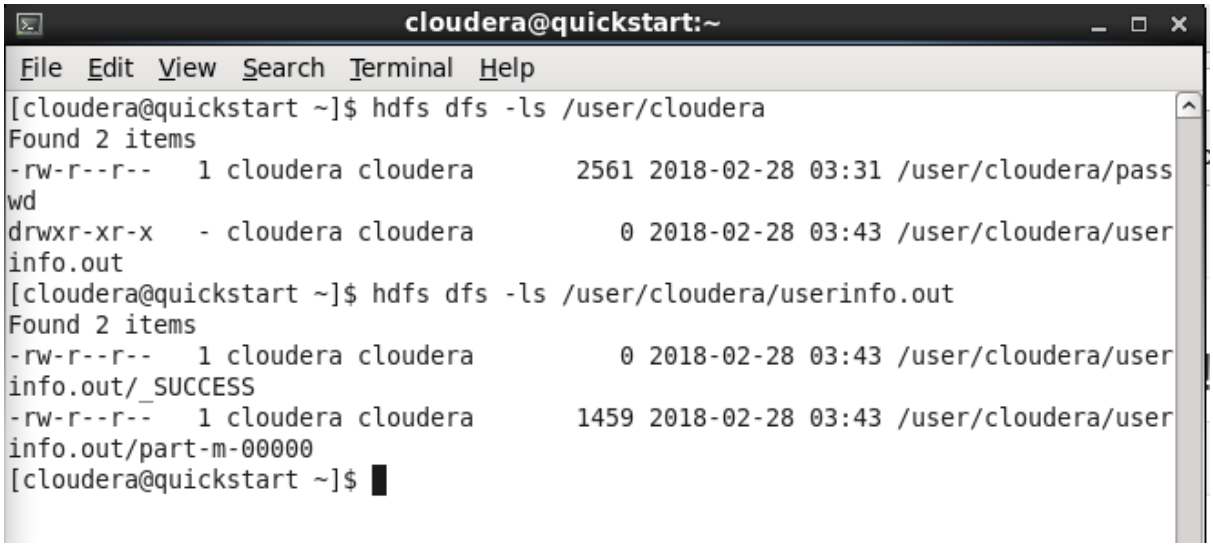


Рисунок 19 – Проверка наличия новой информации в HDFS

3.7. Использовать возможности Hive.

Начните с загрузки файла в HDFS:
hdfs dfs -put /etc/passwd /tmp/
Запустите beeline для интерактивного доступа:
beeline -u jdbc:hive2://

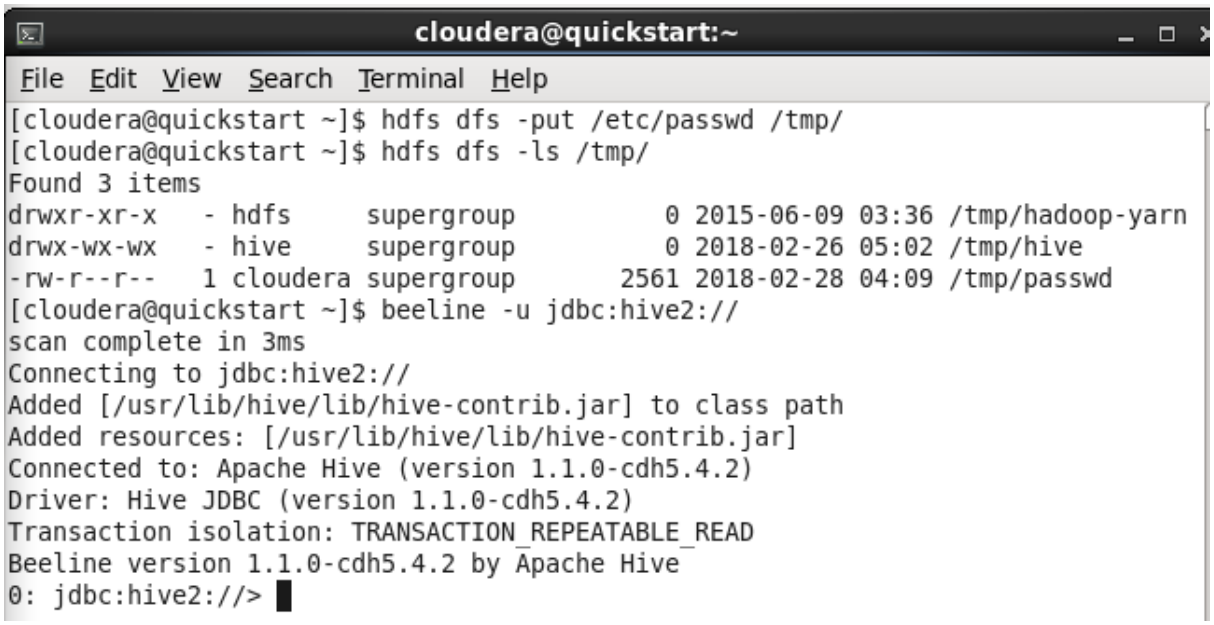


Рисунок 20 – Начало работы с Hive

Выполните следующие команды:

- создание таблицы:

CREATE TABLE userinfo (uname STRING, pswd STRING, uid INT, gid INT, fullname STRING, hdir STRING, shell STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY ':' STORED AS TEXTFILE;

- загрузка файла с паролем из HDFS:

LOAD DATA INPATH '/tmp/passwd' OVERWRITE INTO TABLE userinfo;

- вывод информации:

SELECT uname, fullname, hdir FROM userinfo ORDER BY uname;

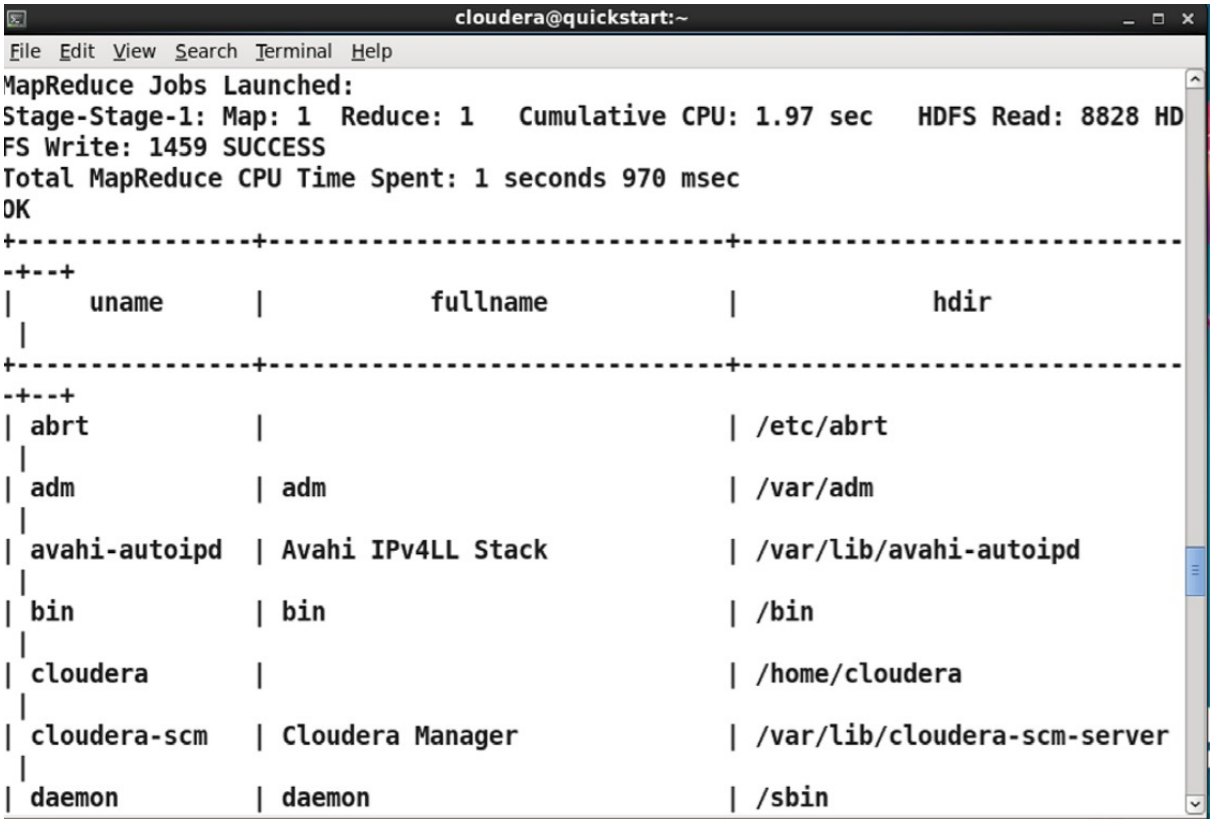


Рисунок 21 – Результаты работы утилиты Hive

Обратите внимание, что Select Info запускает Hadoop job и выводит результаты ее работы.

3.8. Иследовать возможности HBase.

Запуск HBase shell: hbase shell

Создание таблицы:

create ‘usertableinfo’,{NAME=>’username’},{NAME=>’fullname’}, {NAME=>’hom edir’}

Добавление произвольных данных в таблицу usertableinfo

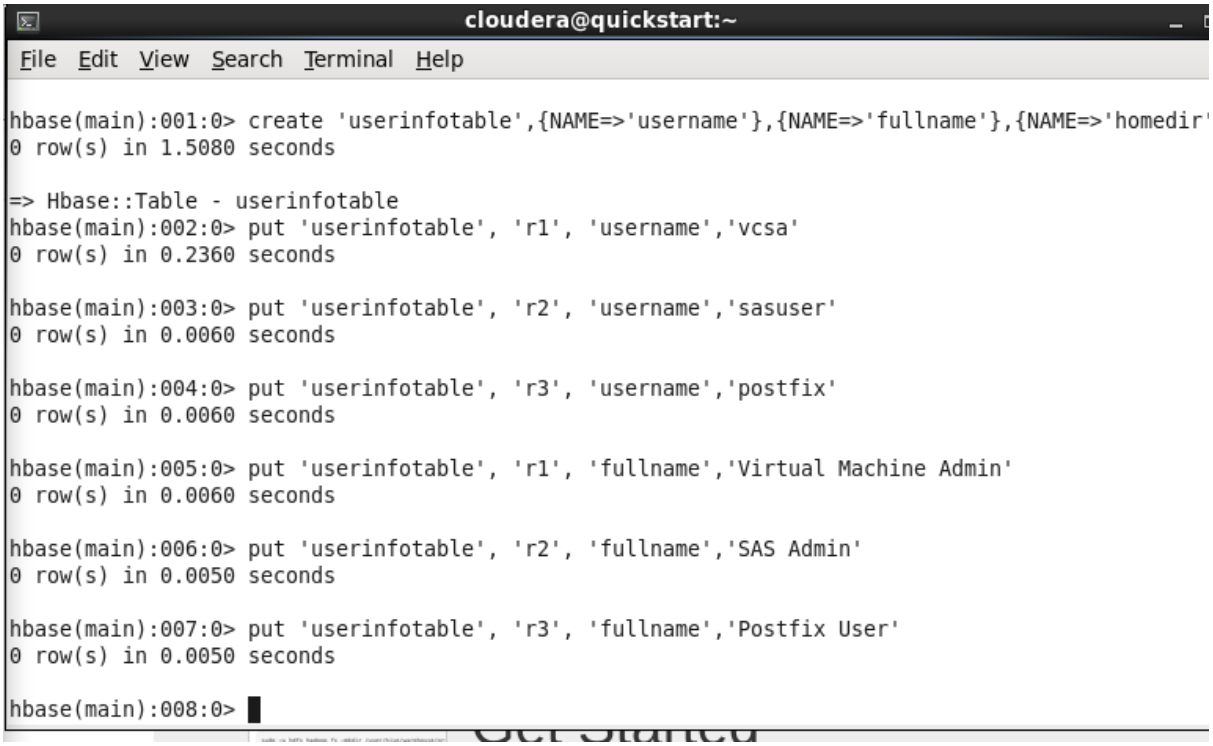


Рисунок 22 – Добавление данных в таблицу с помощью HBase

Просмотр содержимого таблицы:

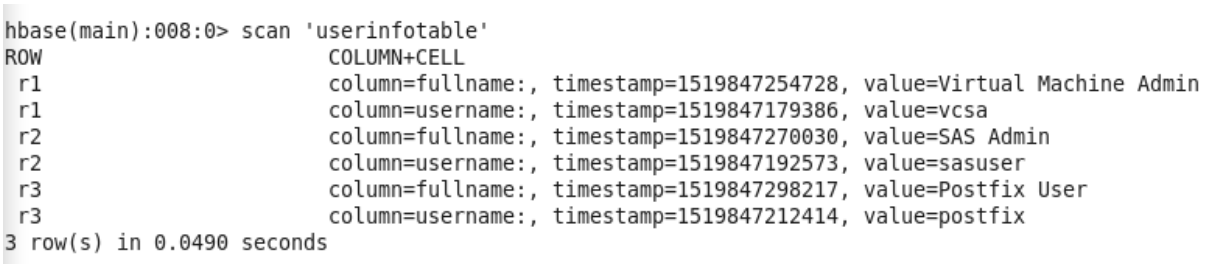


Рисунок 23 – Просмотр содержимого таблицы HBase

Также HBase позволяет выполнять просмотр данных, соответствующих некоторому фильтру:
hbase(main):079:0> scan ‘sales_fact’, { FILTER => “KeyOnlyFilter()”} –> Returns Key
hbase(main):080:0> scan ‘sales_fact’, { FILTER => “FirstKeyOnlyFilter()”} –> Returns Key

4. ЗАДАНИЕ НА РАБОТУ

4.1. Выполнить пункты 3.1 – 3.5 методических указаний, затем

реализовать запрос на выборку данных в соответствии с вариантом задания, приведённом в таблице 1.

4.2. Выполнить пункты 3.6 – 3.8 методических указаний.

4.3. Используя HBase, выполнить создание таблицы, реализовать и исследовать запросы в соответствии с вариантом задания (поля таблицы можно составить произвольно).

Таблица 1 – Варианты заданий

Номер задания	Вариант	Таблица	Условие вывода
1	1	orders	Все заказы со статусом «Complete»
	2	customers	Все из города Ялта
	3	categories	Все из отдела №6
	4	products	Все продукты с ценой меньше 100 рублей
2	1	categories	Вывести с использованием функции scan
	2	products	
	3	orders	
	4	customers	

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

5.1. Как расшифровывается SQOOP?
5.2. Назовите части базового стека Hadoop «Zoo».
5.3. Что изменилось в Hadoop 2.0 по сравнению с Hadoop 1.0?
5.4. Объясните отличие схемы данных Sqoop от таблицы.
5.5. Что такое Apache Hive и в чем его назначение?
5.6. Назовите области применения Hive.
5.7. Что такое Apache HBase?
5.8. Обоснуйте отличие синтаксиса HBase от стандартного SQL.
5.9. Назовите 2 основных компонента уровня MapReduce.

5.10. Назовите преимущества Hadoop по сравнению с традиционными платформами хранения и обработки данных.

ЛАБОРАТОРНАЯ РАБОТА №2
ИССЛЕДОВАНИЕ СПОСОБОВ ИСПОЛЬЗОВАНИЯ РАСПРЕДЕЛЕННОЙ ФАЙЛОВОЙ СИСТЕМЫ HDFS

1. ЦЕЛЬ РАБОТЫ

Изучить способы доступа к распределенной файловой системе HDFS. Получить практические навыки взаимодействия с файловой системой HDFS. Исследовать эффективность различных способов работы с распределенной файловой системой.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

HDFS (Hadoop Distributed File System) – файловая система, предназначенная для хранения файлов больших размеров, поблочно распределённых между узлами вычислительного кластера. HDFS обеспечивает высокопроизводительный доступ к данным приложения и подходит для приложений с большими наборами данных.

Все блоки в HDFS (кроме последнего блока файла) имеют одинаковый размер, и каждый блок может быть размещен на нескольких узлах, размер блока и коэффициент репликации (количество узлов, на которых должен быть размещён каждый блок) определяются в настройках на уровне файла.

Схематично расположение блоков данных на дисках узлов вычислительного кластера показано на рисунке 1.

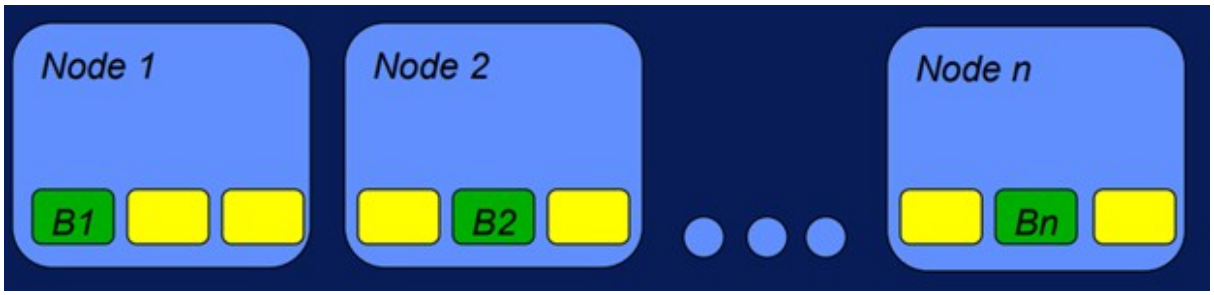


Рисунок 1 – Схематичное изображение расположения блоков файлов относительно узлов вычислительного кластера

2.1. Архитектура HDFS

Развёртывание экземпляра HDFS предусматривает наличие центрального узла имён (англ. name node), хранящего метаданные файловой системы и метаинформацию о распределении блоков, главного сервера, который управляет пространством имен файловой системы и регулирует доступ к файлам клиентами и серии узлов данных (англ. data node), непосредственно хранящих блоки файлов. Узел имён отвечает за обработку операций уровня файлов и каталогов – открытие и закрытие файлов, манипуляция с каталогами, узлы данных непосредственно отрабатывают операции по записи и чтению данных. Узел имён и узлы данных снабжаются веб-серверами, отображающими текущий статус узлов и позволяющими просматривать содержимое файловой системы.

Схематично архитектура HDFS показана на рисунке 2.

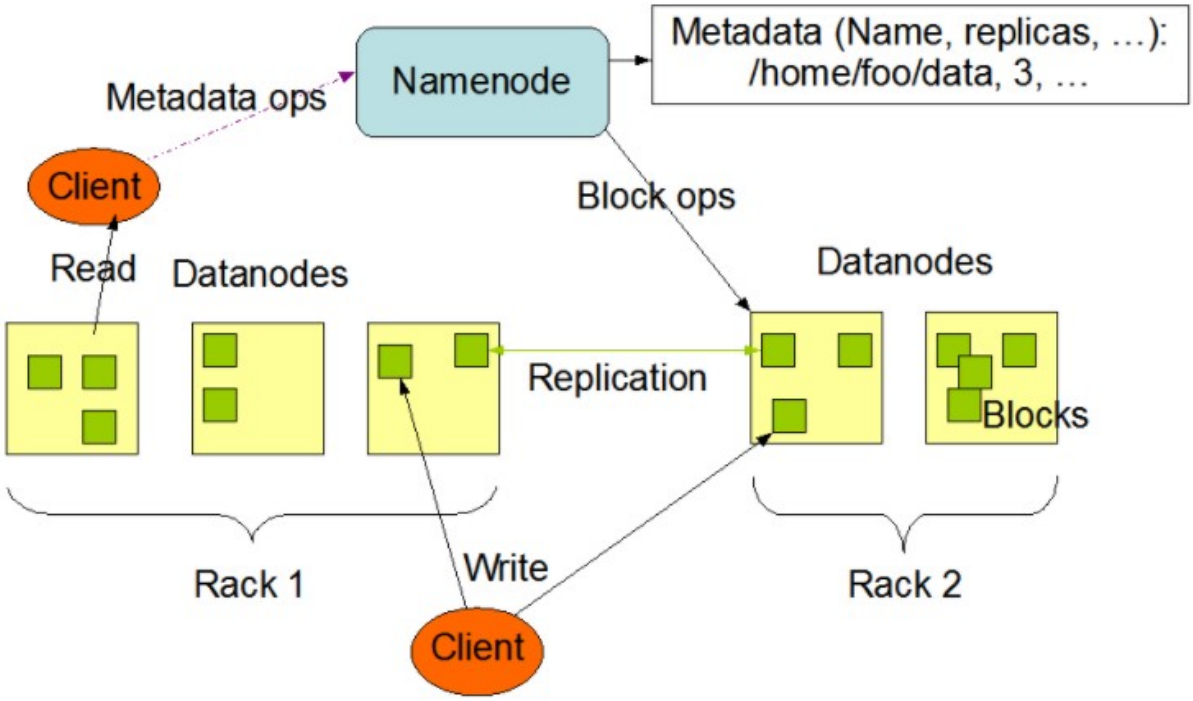


Рисунок 2 – Схематичное изображение архитектуры HDFS

HDFS разработан с использованием языка Java. Любой компьютер, который поддерживает Java, может запускать NameNode или DataNode. Типичный сценарий развертывания представляет собой выделенный компьютер, на котором выполняется только программное обеспечение NameNode и остальные компьютеры в кластере, которые запускают один экземпляр программного обеспечения DataNode. Архитектура не исключает запуск нескольких DataNodes на одном компьютере.

Наличие единственного NameNode в кластере значительно упрощает архитектуру системы. NameNode - это репозиторий для всех метаданных HDFS. Система разработана таким образом, что пользовательские данные никогда не проходят через NameNode.

HDFS поддерживает традиционную иерархическую файловую организацию. Пользователь или приложение могут создавать каталоги и хранить файлы внутри этих каталогов. Иерархия пространства имен файловой системы похожа на большинство других существующих файловых систем; можно создавать и удалять файлы, перемещать файл из одного каталога в другой или переименовывать файл. HDFS поддерживает квоты пользователей и разрешения доступа. HDFS не поддерживает жесткие ссылки или программные ссылки. Однако архитектура HDFS не исключает возможности реализации этих функций.

2.2. Сервер метаданных NameNode

Пространство имен файловой системы HDFS представлено иерархией файлов и директорий. Файлы и директории представлены структурами inode на сервере метаданных (NameNode). Содержимое файла разделяется на большие блоки (обычно размером в **128** мегабайт, но этот размер также может задаваться пользователем для каждого из файлов) и каждый такой блок файла независимо копируется на множество серверов данных приложений. Текущая архитектура предусматривает возможность использования одного сервера метаданных (NameNode) для каждого кластера. Кластер может содержать тысячи серверов данных приложений и обслуживать десятки тысяч клиентов HDFS, так как каждый сервер данных приложений позволяет выполнять множество приложений одновременно.

2.3. Сервер данных приложений DataNode

Каждая копия блока файла на сервере данных приложений представлена двумя файлами в локальной файловой системе. Первый файл содержит сами данные, а второй - метаданные блока, включающие в себя контрольные суммы для блока данных и метку времени, относящуюся к моменту генерации блока. Размер файла данных равен используемому размеру блока и файл не занимает дополнительного дискового пространства

для округления в сторону повышения размера файла до размера блока таким образом, как это делается в традиционных файловых системах. Следовательно, если размер блока составляет половину установленного размера, он занимает объем локального диска, соответствующий только половине установленного размера блока.

Во время загрузки каждый сервер данных приложений соединяется с сервером метаданных и осуществляет операцию рукопожатия. Целью этой операции рукопожатия является проверка идентификатора пространства имен и версии программного обеспечения на сервере данных приложений. Если хотя бы одно из используемых сервером метаданных значений не совпадает со значением, используемым сервером данных приложений, сервер данных приложений автоматически прекращает свою работу.

Благодаря репликации обеспечивается устойчивость распределенной системы к отказам отдельных узлов. Файлы в HDFS могут быть записаны лишь однажды (модификация не поддерживается), а запись в файл в одно время может вести только один процесс. Организация файлов в пространстве имён — традиционная иерархическая: есть корневой каталог, поддерживается вложение каталогов, в одном каталоге могут располагаться и файлы, и другие каталоги.

2.4. Способы взаимодействия с HDFS

HDFS поддерживает несколько способов взаимодействия:

- REST API;
- Доступ через командную строку (hdfs dfs [COMMAND [COMMAND_OPTIONS]]);
- Доступ через Java API.

2.4.1. Список основных команд HDFS

Рассмотрим базовые команды, доступные с помощью bin/hadoop dfs. Запуск bin/hadoop dfs без дополнительных аргументов выведет список всех команд, которые могут быть запущены системой FsShell. Кроме того, bin/hadoop dfs -help <commandName> покажет краткую подсказку для любой операции.

Список команд приведен в таблице 1. Используются следующие обозначения параметров:

- курсивом обозначены переменные, заполняемые пользователем;
- «path» означает имя файла или папки;
- «path...» означает одно и более имен файлов или папок;
- «file» означает любое имя файла;
- «src» и «dest» – пути к файлам, используемые при выполнении операций в HDFS;
- «localSrc» и «localDest» – те же пути, только в локальной файловой системе. Все остальные имена путей и файлов относятся к объектам внутри HDFS.
- параметры в [квадратных скобках] опциональны.

Таблица 1 – Команды взаимодействия с HDFS

Команда	Операция
-ls *path*	Выводит содержимое директории по указанному пути, показывая названия, разрешения, владельца, размер и дату изменения каждой записи.
-lsr *path*	Ведет себя как -ls, но рекурсивно отображает записи во всех подкатегориях пути.
-du *path*	Показывает использование диска в байтах для всех файлов, с подходящим path.
-dus *path*	Работает как -du, но отображает суммарную занятость диска всеми файлами/каталогами в указанном path
-mv *src dest*	Перемещает файл или каталог, отмеченный *src* в *dest* в HDFS.
-cp *src dest*	Копирует файл или каталог, отмеченный *src* в *dest* в HDFS.
-rm *path*	Удаляет файл или пустой каталог, выявленный *path*.
-rmr *path*	Удаляет файл или каталог, выявленный *path*. Рекурсивно удаляет все дочерние записи (например, файлы или подкаталоги path).
-put *localSrcdest*	Копирует все файлы или каталоги из локальной файловой системы, отмеченные *localSrc* в *dest* по DFS.

Продолжение таблицы 1

Команда	Операция
-copyFromLocal *localSrc dest*	Идентичен -put
-moveFromLocal *localSrc dest*	Копирует файлы или каталоги из локальной файловой системы, отмеченные *localSrc* в *dest* в HDFS, затем, в случае успеха, удаляет локальную копию.
-get [-crc] *srclocalDest*	Копирует файл или каталог в HDFS, выявленные *src* по пути локальной файловой системы, определенному *localDest*.
-getmerge *srclocalDest*[addnl]	Получает все файлы, которые совпадают с *src* в HDFS, и копирует их в отдельный общий файл, определенным *localDest*.
-cat *filename*	Отображает содержимое *filename* на стандартный вывод.
-copyToLocal [-crc] *srclocalDest*	Идентичен -get
-moveToLocal [-crc] *srclocalDest*	Работает как -get, но в случае успеха удаляет копии HDFS.
-mkdir *path*	Создает каталог по имени *path* в HDFS. Создает любые дочерние папки в *path* , которые утеряны (например, mkdir -p в Linux).
-setrep [-R] [-w]*rep path*	Устанавливает коэффициент целевой репликации для файлов из *path* в *rep*.
-touchz *path*	Создает файл по *path* , сохраняя текущую страницу как timestamp. Выдает ошибку, если файл уже существует в пути или файл уже имеет значение 0.
-test -[ezd] *path*	Возвращает 1, если *path* существует; имеет нулевую длину; или каталог или 0.

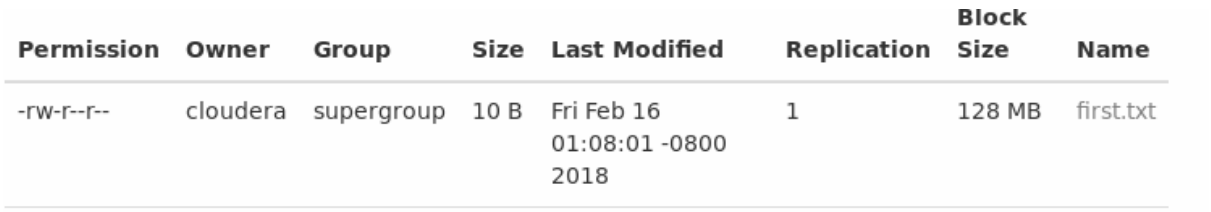
Продолжение таблицы 1

Команда	Операция
-stat [format]*path*	Выводит данные о *path*. *Format* - строка, принимающая размер файла в блоках (%b), имени файла (%n), размере блока (%o), копировании (%r) и дате изменения. (%y, %Y).
-tail [-f] *file*	Показывает последний 1KB *file* при стандартном выводе.
-chmod [-R]*mode,mode,...path...*	Изменяет права доступа к файлам связанных с одним или несколькими объектами, идентифицированными по path. Выполняет изменения рекурсивно с флагом -R, 3-значный восьмеричный режим или {augo} +/- {rwxX}. Предполагается, что не указана область и не применяется umask.
-chown [-R] [*owner*][:[*group*]] *path...*	Устанавливает пользователя-владельца или группу владельцев для файлов и каталогов, определенных в пути.... Выставляет владельца рекурсивно, если определен -R .
-chgrp [-R]*group path...*	Устанавливает группу владельцев для файлов или каталогов, определенных *path*.... Устанавливает группу рекурсивно, если определен -R .
-help *cmd*	Возвращает использование информации для одной из команд выше. Используйте знак '-' в *cmd.*

Для примера создадим текстовый файл first.txt и добавим в него произвольный текст, после чего скопируем этот файл из из локальной файловой системы в HDFS. Можем посмотреть содержимое файла с

помощью команды hadoop fs -cat или в NameNode Web UI cloudera Manager (рисунок 3). Для удаления созданного файла в корзину воспользуемся командой hadoop fs -rm, для полного удаления файла из HDFS необходимо добавить флаг -skipTrash. Пример сценария приведен ниже:

```
[cloudera@quickstart ~]$ echo 'test hdfs' >> first.txt
[cloudera@quickstart ~]$ hadoop fs -put first.txt /user/root/
[cloudera@quickstart ~]$ hadoop fs -cat /user/root/first.txt
test hdfs
[cloudera@quickstart ~]$ hadoop fs -cp /user/root/first.txt /user/root/second.txt
[cloudera@quickstart ~]$ hadoop fs -ls /user/root/
Found 2 items
-rw-r--r--   1 cloudera supergroup         10 2018-02-16 01:36 /user/root/first.txt
-rw-r--r--   1 cloudera supergroup         10 2018-02-16 01:36 /user/root/second.txt
[cloudera@quickstart ~]$ hadoop fs -rm -skipTrash /user/root/second.txt
Deleted /user/root/second.txt
[cloudera@quickstart ~]$ hadoop fs -ls /user/root/
Found 1 items
-rw-r--r--   1 cloudera supergroup         10 2018-02-16 01:36 /user/root/first.txt
```



Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
-rw-r--r--	cloudera	supergroup	10 B	Fri Feb 16 01:08:01 -0800 2018	1	128 MB	first.txt

Рисунок 3 – Отображение файлов в NameNode Web UI

2.4.2. Java API

В виртуальной машине Quickstart VM есть всё необходимое для разработки приложения на Java использующих HDFS Java API, а именно IDE Eclipse, JDK, JRE, и набор библиотек Hadoop, расположенный по адресу /usr/lib/Hadoop/client/.

После создания проекта в Eclipse необходимо подключить вышеуказанный набор библиотек, а также создать файл *log4j.properties* по адресу {ProjectRoot}/conf/ со следующим содержимым:

```
hadoop.root.logger=DEBUG, console
```

```
log4j.rootLogger = DEBUG, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p
%c{2}: %m%n
```

Это настройки Hadoop logging framework, а именно вывода логов в консоль в поток system.err.

Прежде чем приступить к написанию кода, следует понимать, что Java API понятия не имеет где находится HDFS. Адрес текущей файловой системы можно определить с помощью команды *hdfs getconf -confKey fs.defaultFS.*

Пример кода на языке Java, с использованием Java API:

```
package lab2;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;

public class lab2 {

        public static void main(String[] args) throws Exception {
                System.out.printf("BDIOT LR2\n");
                String uri = "/tmp/lab2/hosts";
                String folderuri = "/tmp/lab2/fldr";
                Configuration conf = new Configuration();

                conf.set("fs.defaultFS", "hdfs://localhost:8020");
    System.out.printf("[FileSystem.get(conf)]\n");

                FileSystem fs = FileSystem.get(conf);
                System.out.printf("[fs.open(new Path(uri))]\n");
                FSDataInputStream in = fs.open(new Path(uri));

                System.out.printf("10 bytes output of "+uri+" :\n");
                byte[] inbuf = new byte[100];
                in.read(inbuf, 0, 100);
                System.out.write(inbuf);
```

```
                System.out.printf("\n[in.close()]\n");
                in.close();
                }
        }
}
```

Данный код считывает 10 байт данных из файла, находящегося в HDFS, и печатает содержимое на экран. Для начала работы необходимо инстанциировать объект Configuration с данными об адресе HDFS, создать объект FileSystem, передав в качестве аргумента конструктора объект Configuration, и создать файловый поток. Далее работа с этим файловым потоком не отличается от обычной в языке Java.

2.4.3. Использование REST API

Для взаимодействия с HDFS через REST API можно использовать любой удобный инструмента, поддерживающий HTTP, например утилиту curl, как показано на рисунке 4. Результаты выполнения команд возвращаются в формате JSON.

```
[root@quickstart cloudera]# curl -i "http://quickstart.cloudera:14000/webhdfs/v1
/tmp/lab2/fldr/testfile?user.name=cloudera&op=GETFILESTATUS"
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: hadoop.auth="u=cloudera&p=cloudera&t=simple&e=1510217245383&s=i1dc5x
LomVQXUP1/9jSx1+sJ0tc="; Path=/; Expires=Thu, 09-Nov-2017 08:47:25 GMT; HttpOnly
Content-Type: application/json
Transfer-Encoding: chunked
Date: Wed, 08 Nov 2017 22:47:56 GMT

{"FileStatus":{"pathSuffix":"","type":"FILE","length":14,"owner":"cloudera","gro
up":"supergroup","permission":"644","accessTime":1510180845535,"modificationTime
":1510180846888,"blockSize":134217728,"replication":3}}
[root@quickstart cloudera]# curl -i "http://quickstart.cloudera:14000/webhdfs/v1
/tmp/lab2/fldr/testfile?user.name=cloudera&op=OPEN"
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: hadoop.auth="u=cloudera&p=cloudera&t=simple&e=1510217402937&s=nJKNfV
7DON2G3Wy8p84jtIv1QeQ="; Path=/; Expires=Thu, 09-Nov-2017 08:50:02 GMT; HttpOnly
Content-Type: application/octet-stream
Content-Length: 14
Date: Wed, 08 Nov 2017 22:50:05 GMT


Hello world![root@quickstart cloudera]#
```

Рисунок 4 – Взаимодействие с WEBHDFS REST API с помощью curl

Для взаимодействия с HDFS через REST API можно реализовать и собственный HTTP-клиент. Пример реализации на языке C# приведён ниже:

```
using System;
using System.Net.Http;
using System.Net.Http.Headers;

namespace restapil2
{
        class Program
   {
      private const string URL =
"http://192.168.153.130:14000/webhdfs/v1/tmp/lab2/hosts";
      private const string urlParameters = "?user.name=cloudera&op=OPEN";

      static void Main(string[] args)
      {
         HttpClient client = new HttpClient();
         client.BaseAddress = new Uri(URL);

         // Add an Accept header for JSON format.
         client.DefaultRequestHeaders.Accept.Add(
         new MediaTypeWithQualityHeaderValue("application/json"));

         // List data response.
         HttpResponseMessage response = client.GetAsync(urlParameters).Result;
         if (response.IsSuccessStatusCode)
         {
               var dataObjects = response.Content.ReadAsStringAsync();
            Console.WriteLine(dataObjects.Result);
         }
         else
         {
               Console.WriteLine("{0} ({1})", (int)response.StatusCode,
response.ReasonPhrase);
         }
         Console.WriteLine("Press any key...");
         Console.ReadKey();
        }
```

```
    }
}
```

Эта программа осуществляет считывание содержимого файла (переменная URL) с помощью команды “OPEN” и вывод содержимого на экран.

3. ЗАДАНИЕ НА РАБОТУ

3.1. Создайте тестовый подкаталог внутри файловой системы HDFS.

3.2. С помощью команды copyFromLocal переместите какой-либо файл из локальной файловой системы в вышеупомянутый подкаталог HDFS.

3.3. Для дополнительной уверенности просмотрите перемещенный файл в файловой системе HDFS с помощью команды hdfs dfs .

Сигнатура команды:

hdfs dfs -copyFromLocal <local_FS_filename> <target_on_HDFS>

3.4.После перемещения с помощью интерфейса Java API напишите приложение, считывающее данные этого файла из HDFS и выводящее содержимое файла на экран;

3.5. Используя curl, осуществите обращение к HDFS с помощью REST API, которое выведет содержимое созданного вами тестового файла;

3.6. Используя REST API, напишите на любом языке программирования метод, который осуществляет обращение к файловой системе HDFS и выводит содержимое файла на экран.

3.7. Проведите сравнительное исследование скорости взаимодействия с HDFS различными способами. Результаты отразите в отчете.

3.7. Осуществите проверку целостности файловой системы с помощью команды hdfs fsck.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Что такое HDFS? Перечислите основные собенности HDFS.

4.2. Каковы два основных типа узлов в HDFS? Для чего они используются?

4.3. Каковы два основных уровня архитектуры Hadoop?

4.4. Какие существуют виды взаимодействия с HDFS?

4.5. Как происходит процесс записи/чтения в HDFS?

4.6. С помощью какой команды можно осуществить проверку целостности файловой системы?

4.7. Приведите пример отображения статуса файла с помощью REST API.

4.8. На какое количестве блоков разделится содержимое файла размером 10 гигабайт при стандартных настройках?

4.9. Назовите основные преимущества Hadoop по сравнению с традиционными файловыми системами.

4.10. Какие новые функции были добавлены в Hadoop 2.0 HDFS2?

4.11. Что такое HDFS Snapshots? Перечислите основные команды для работы с Snapshots.

3. ЛАБОРАТОРНАЯ РАБОТА №3
ИССЛЕДОВАНИЕ СПОСОБОВ ВЫБОРКИ ДАННЫХ НА ОСНОВЕ MAP/REDUCE

1. ЦЕЛЬ РАБОТЫ

Исследовать основные возможности выборки данных на основе MapReduce. Получить практические навыки применения MapReduce для решения задач анализа текстовых файлов.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Понятие MapReduce

MapReduce – это подход, алгоритм или паттерн параллельной обработки больших объемов данных, например логов веб-запросов. Существуют разные реализации MapReduce. Достаточно известна и запатентована реализация этого алгоритма и подхода Google.

В общем случае MapReduce – это фреймворк для вычисления некоторых наборов распределенных задач с использованием большого количества компьютеров (называемых «нодами»), образующих кластер.

2.2. Алгоритм работы MapReduce

Алгоритм получает на вход 3 аргумента: исходную коллекцию, Map функцию, Reduce функцию, и возвращает новую коллекцию данных:

Collection MapReduce(Collection source, Function map, Function reduce)

Алгоритм состоит из нескольких шагов. В качестве первого шага примеряется функция Map к каждому элементу исходной коллекции. Map вернет ноль либо создаст экземпляры Key/Value-объектов:

ArrayOfKeyValue Map(object itemFromSourceCollection)

Таким образом, обязанность функции Map – конвертировать элементы исходной коллекции в ноль или несколько экземпляров Key/Value объектов. Это продемонстрировано ниже, на рисунке 1.

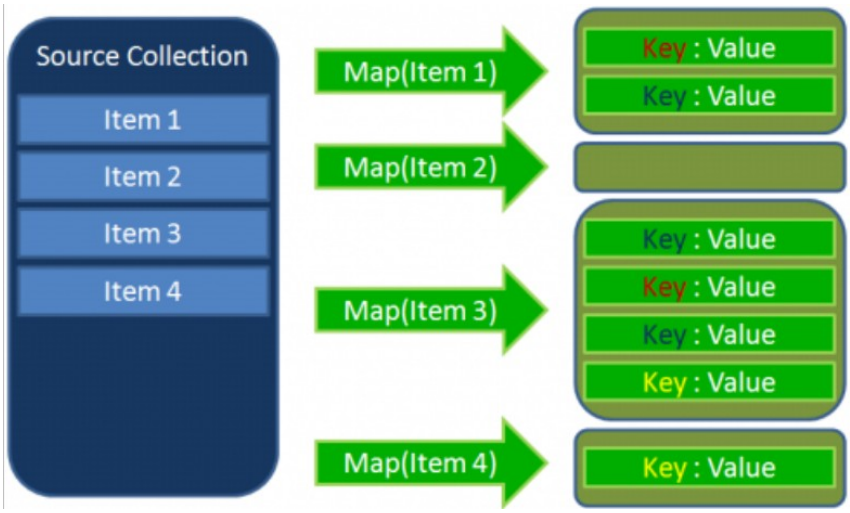


Рисунок 1 – Принцип работы Map

Следующим шагом алгоритм сортирует пары Key/Value по ключу и создает новые экземпляры объектов с группировкой value по ключу (рисунок 2).

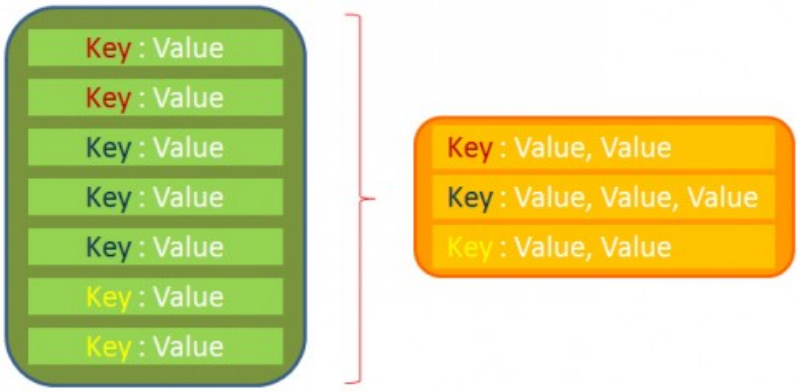


Рисунок 2 – Группировка значений по ключу в Map

Последним шагом выполняется функция Reduce для каждого сгруппированного экземпляра Key/Value объекта:

ItemResult Reduce(KeyWithArrayOfValues item)

Функция Reduce вернет новый экземпляр объекта, который будет включен в результирующую коллекцию (рисунок 3).

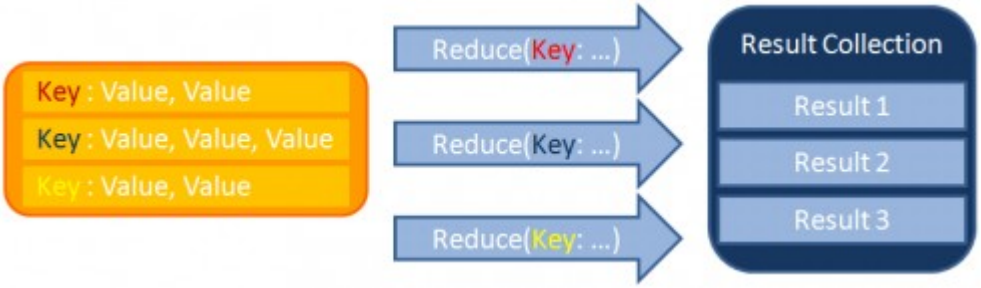


Рисунок 3 – Принцип работы Reduce

2.3. Требования к производительности MapReduce

Не гарантируется, что программы MapReduce будут выполняться быстрыми темпами. Основное преимущество этой модели программирования заключается в том, чтобы использовать оптимизированную операцию перемешивания платформы и только написать map и reduce части программы. На практике автор программы MapReduce, однако, должен принять во внимание шаг перемешивания; в частности, частная функция и объем данных, записанных функцией Map, могут иметь большое влияние на производительность и масштабируемость. Дополнительные модули, такие как функция Combiner, могут уменьшить объем данных, записываемых на диск и передающихся по сети.

При разработке алгоритма MapReduce автору необходимо выбрать хороший компромисс между стоимостью вычислений и стоимостью связей.

Для быстро завершающихся процессов и того, где данные вписываются в основную память одной машины или небольшого кластера, использование среды MapReduce обычно неэффективно. Поскольку эти структуры предназначены для восстановления после потери целых узлов во время вычисления, они записывают промежуточные результаты в распределенное хранилище. Это имеет смысл только тогда, когда вычисление включает в себя множество компьютеров и длительное время выполнения вычислений. Задача, которая завершается за секунды, может быть просто перезапущена в случае ошибки, и вероятность того, что по меньшей мере ошибка произошла на одной машине, быстро растет с размером кластера.

2.4. Преимущества и недостатки Hadoop MapReduce

Преимущества:

1. Эффективная работа с большим (от 100 Гб) объемом данных;
2. Масштабируемость;
3. Отказоустойчивость;
4. Унифицированность подхода;
5. Предоставление разработчику сравнительно «чистой» абстракции;
6. Снижение требований к квалификации разработчика, в том числе его знаний и опыта по написанию многопоточного кода;
7. Дешевизна лицензирования (Open Source).

Ограничения:

1. Смешение ответственности для Reducer (сортировка и агрегация данных). Таким образом, Reducer – это все, что «не map»;

2. Отсутствие контроля над потоком данных у разработчика (поток данных управляется фреймворком Hadoop MapReduce автоматически);

3. Как следствие предыдущего пункта, невозможность простыми средствами организовать взаимодействие между параллельно выполняющимися потоками.

Недостатки:

1. Применение MapReduce по производительности менее эффективно, чем специализированные решения;

2. Эффективность применение MapReduce снижается при малом количество машин в кластере (высоки издержки на взаимодействие, а степень распараллеливания невелика);

3. Невозможно предсказать окончание стадии map;

4. Этап свертки не начинается до окончания стадии map;

5. Как следствие предыдущего пункта, задержки в исполнении любого запущенного map-задания ведут к задержке выполнения задачи целиком;

6. Низкая утилизация ресурсов вследствие жесткого деления ресурсов кластера на map- и reduce-слоты.

7. Сбой узла JobTracker приводит к простою всего кластера.

2.5. Архитектура Hadoop MapReduce

Hadoop MapReduce использует архитектуру «master-worker», где master – единственный экземпляр управляющего процесса (JobTracker), как правило, запущенный на отдельной машине (вычислительном узле). Worker-процессы – это произвольное множество процессов TaskTracker, исполняющихся на DataNode.

JobTracker и TaskTracker «лежат» над уровнем хранения HDFS, и запускаются/исполняются в соответствии со следующими правилами:

- экземпляр JobTracker исполняется на NameNode-узле HDFS;
- экземпляры TaskTracker исполняются на DataNode-узле.

TaskTracker исполняются в соответствии с принципом «данные близко», т.е. процесс TaskTracker располагается топологически максимально близко с узлом DataNode, данные которого обрабатываются.

Вышеописанные принципы расположения JobTracker- и TaskTracker-процессов позволяют существенно сократить объемы передаваемых по сети

данных и сетевые задержки, связанные с передачей этих данных – основные «узкие места» производительности в современных распределенных системах.

JobTracker является единственным узлом, на котором выполняется приложение MapReduce, вызываемое программным клиентом. JobTracker выполняет следующие функции:

- планирование индивидуальных (по отношению к DataNode) заданий map и reduce, промежуточных свёрток;
- координация заданий;
- мониторинг выполнения заданий;
- переназначение завершившихся неудачей заданий другим узлам TaskTracker.

В свою очередь, TaskTracker выполняет следующие функции:

- исполнение map- и reduce-заданий;
- управление исполнением заданий;
- отправка сообщений о статусе задачи и завершении работы узлу JobTracker;
- отправка диагностических heartbeat-сообщений узлу JobTracker.

Взаимодействие TaskTracker-узлов с узлом JobTracker идет посредством RPC-вызовов, причем вызовы идут только от TaskTracker. Аналогичный принцип взаимодействия реализован в HDFS – между узлами DataNode и NameNode-узлом. Такое решение уменьшает зависимость управляющего процесса JobTracker от процессов TaskTracker.

Взаимодействие JobTracker-узла с клиентом (программным) проходит по следующей схеме: JobTracker принимает задание (Job) от клиента и разбивает задание на множество M map-задач и множество R reduce-задач. Узел JobTracker использует информацию о файловых блоках (количество блоков и их месторасположение), расположенную в узле NameNode, находящемуся локально, чтобы решить, сколько подчиненных задач необходимо создать на узлах типа TaskTracker. TaskTracker получает от JobTracker список задач (тасков), загружает код и выполняет его. Периодично TaskTracker отсылает JobTracker статус выполнения задачи.

Взаимодействия TaskTracker-узлов с программным клиентом отсутствуют.

По аналогии с архитектурой HDFS, где NameNode является единичной точкой отказа (Single point of failure), JobTracker также является таковой. Принцип восстановления в узлах JobTracker и TaskTracker описан ниже.

При сбое TaskTracker-узла JobTracker-узел переназначает задания неисправного узла другому узлу TaskTracker. В случае неисправности

JobTracker-узла, для продолжения исполнения MapReduce-приложения, необходим перезапуск JobTracker-узла. При перезапуске узел JobTracker читает из специального журнала данные, о последней успешной контрольной точке (checkpoint), восстанавливает свое состояние на момент записи checkpoint и продолжает работу с места последней контрольной точки.

2.6. Запуск вычислений MapReduce на Hadoop

Самый простой способ запустить MapReduce-программу на Hadoop – воспользоваться streaming-интерфейсом Hadoop. Streaming-интерфейс предполагает, что map и reduce реализованы в виде программ, которые принимают данные с stdin и выдают результат на stdout.

Программа, которая исполняет функцию map называется mapper. Программа, которая выполняет reduce, называется, соответственно, reducer.

Streaming интерфейс предполагает по умолчанию, что одна входящая строка в mapper или reducer соответствует одной входящей записи для map.

Вывод mapper’а попадает на вход reducer’у в виде пар (ключ, значение), при этом все пары соответствующие одному ключу:

1. Гарантированно будут обработаны одним запуском reducer’а;

2. Будут поданы на вход подряд (то есть если один reducer обрабатывает несколько разных ключей – вход будет сгруппирован по ключу).

Теперь запустим streaming-задачу при помощи утилиты YARN, которая служит для запуска и управления различными приложениями (в том числе map-reduce based) на кластере. Hadoop-streaming.jar – это как раз один из примеров такого YARN-приложения.

Дальше идут параметры запуска:

- input – папка с исходными данными на hdfs;

- output – папка на hdfs, куда нужно положить результат;

- file – файлы, которые нужны в процессе работы map-reduce задачи (указываются файлы с мапером и редьюсером);

- mapper – консольная команда, которая будет использоваться для map-стадии;

- reduce – консольная команда которая будет использоваться для reduce-стадии.

Результат работы после успешного выполнения складывается на HDFS в папку, которую мы указали в поле output.

2.7. Пример выполнения вычислений MapReduce

Поместим в рабочую папку /tmp/lab3 (в HDFS) папку с данными data1. В data1 содержится 3 файла file1-file3. Выведем содержимое file1. Все перечисленное выше изображено на рисунке 4.

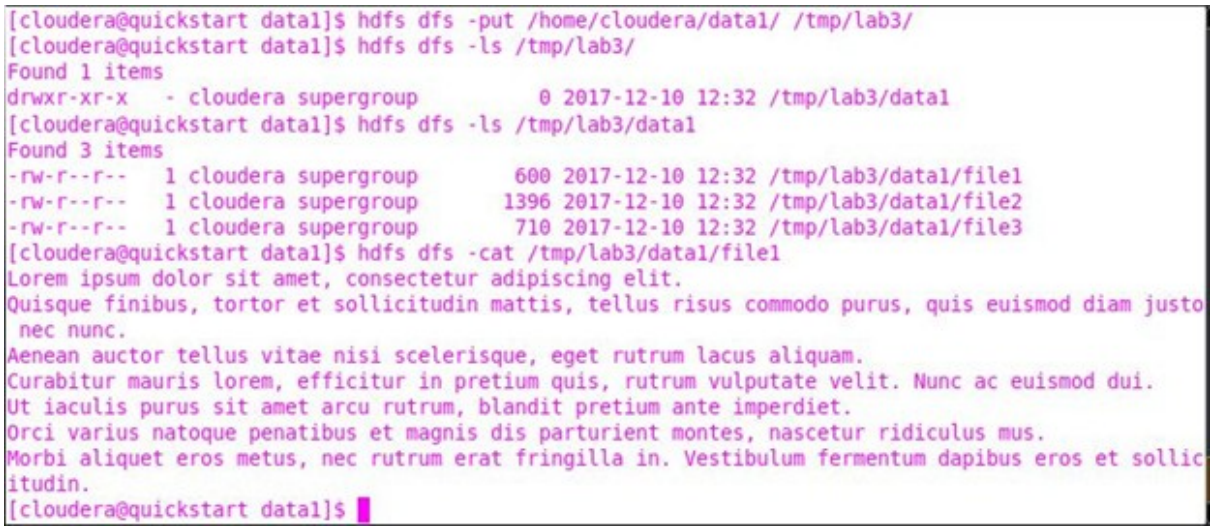


Рисунок 4 – Размещение файлов и просмотр содержимого одного из них

Еще один вариант – по примеру лабораторной работы №2 создать файлы, ввести их содержимое и добавить их в HDFS.

Составим mapper:

```
#!/usr/bin/env python
import sys

for line in sys.stdin:
    line = line.strip()
    line = "".join(line.split())
    for bukva in line:
    print('%s\t%s' % (bukva, 1))
```

Данный mapper берет из стандартного потока *stdin* строки, пока они не кончатся, и выдает для каждого символа данной строки кортеж (<*буква*>,1).

Составим reducer:

```
#!/usr/bin/env python
import sys

current_char = None
```

```
current_count = 0
ch = None

for line in sys.stdin:
        line = line.strip()
        ch, count = line.split('\t', 1)

        try:
        count = int(count)
        except ValueError:
        continue

        if current_char == ch:
        current_count += count
        else:
        if current_char:
          print('%s\t%s' % (current_char, current_count))
        current_count = count
        current_char = ch

if current_char == ch:
        print ('%s\t%s' % (current_char, current_count))
```

Данный reducer получает из стандартного потока *stdin* определенное количество строк с кортежами (<*буква*>, 1), при этом строки отсортированы по ключу <*буква*> по возрастанию. Учитывая, что строки отсортированы, просуммировать значения – количество букв, для одинаковых ключей – не проблема. На выходе программы будет список кортежей (<*буква*>,<*количество*>).

Разместим написанные mapper и reducer по адресу /home/cloudera/. Также надо не забыть сделать созданные файлы исполняемыми, с помощью команды chmod –x <*file*>. Запустим написанные mapper и reducer. Результаты запуска показаны на рисунках 5 и 6.

```
[cloudera@quickstart data1]$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar -input /tmp/
lab3/data1/ -output /tmp/lab3/output1/ -mapper /home/cloudera/tm.py -reducer /home/cloudera/tr.py
packageJarJar: [] [/usr/jars/hadoop-streaming-2.6.0-cdh5.4.2.jar] /tmp/streamjob1182804476804585235
.jar tmpDir=null
17/12/10 13:36:08 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
17/12/10 13:36:09 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
17/12/10 13:36:09 INFO mapred.FileInputFormat: Total input paths to process : 3
17/12/10 13:36:09 INFO mapreduce.JobSubmitter: number of splits:3
17/12/10 13:36:09 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1512848733301_0002
17/12/10 13:36:10 INFO impl.YarnClientImpl: Submitted application application_1512848733301_0002
17/12/10 13:36:10 INFO mapreduce.Job: The url to track the job: http://quickstart.cloudera:8088/pro
xy/application_1512848733301_0002/
17/12/10 13:36:10 INFO mapreduce.Job: Running job: job_1512848733301_0002
17/12/10 13:36:17 INFO mapreduce.Job: Job job_1512848733301_0002 running in uber mode : false
17/12/10 13:36:17 INFO mapreduce.Job:  map 0% reduce 0%
17/12/10 13:36:24 INFO mapreduce.Job:  map 33% reduce 0%
17/12/10 13:36:32 INFO mapreduce.Job:  map 100% reduce 0%
17/12/10 13:36:42 INFO mapreduce.Job:  map 100% reduce 100%
17/12/10 13:36:43 INFO mapreduce.Job: Job job_1512848733301_0002 completed successfully
17/12/10 13:36:43 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=13842
                FILE: Number of bytes written=477301
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=3018
                HDFS: Number of bytes written=186
                HDFS: Number of read operations=12
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=3
                Launched reduce tasks=1
                Data-local map tasks=3
                Total time spent by all maps in occupied slots (ms)=29958
                Total time spent by all reduces in occupied slots (ms)=13313
                Total time spent by all map tasks (ms)=29958
                Total vcore-seconds taken by all map tasks=29958
                Total vcore-seconds taken by all reduce tasks=13313
                Total megabyte-seconds taken by all map tasks=30676992
                Total megabyte-seconds taken by all reduce tasks=13632512
        Map-Reduce Framework
                Map input records=23
                Map output records=2306
                Map output bytes=9224
                Map output materialized bytes=13854
                Input split bytes=312
                Combine input records=0
                Combine output records=0
                Reduce input groups=38
                Reduce shuffle bytes=13854
                Reduce input records=2306
                Reduce output records=38
                Spilled Records=4612
                Shuffled Maps =3
                Failed Shuffles=0
                Merged Map outputs=3
                GC time elapsed (ms)=203
                CPU time spent (ms)=3830
                Physical memory (bytes) snapshot=982974464
                Virtual memory (bytes) snapshot=6297473024
                Total committed heap usage (bytes)=772276224
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=2706
        File Output Format Counters
                Bytes Written=186
17/12/10 13:36:43 INFO streaming.StreamJob: Output directory: /tmp/lab3/output1/
[cloudera@quickstart data1]$
```

Рисунок 5 – Запуск созданных mapper и reducer

```
[cloudera@quickstart data1]$ hdfs dfs -ls /tmp/lab3/output1
Found 2 items
-rw-r--r--   1 cloudera supergroup          0 2017-12-10 13:36 /tmp/lab3/output1/_SUCCESS
-rw-r--r--   1 cloudera supergroup        186 2017-12-10 13:36 /tmp/lab3/output1/part-00000
[cloudera@quickstart data1]$ hdfs dfs -cat /tmp/lab3/output1/part-00000
,       37
.       48
;       1
A       5
C       6
D       1
F       1
I       5
M       6
L       2
O       1
N       6
Q       2
P       5
U       4
V       5
a       170
c       102
b       26
e       226
d       55
g       20
f       18
i       229
h       12
j       1
m       109
l       134
o       104
n       131
q       28
p       57
s       172
r       145
u       213
t       183
v       33
x       3
```

Рисунок 6 – Результаты выполнения mapper и reducer

Итак, получилось 3 mapper-а и 1 reducer. Теперь увеличим количество reducer-ов до 2. Процесс запуска команды показан на рисунке 7.

```
[cloudera@quickstart data1]$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-streaming.jar -jobconf map
red.reduce.tasks=2 -input /tmp/lab3/data1/ -output /tmp/lab3/output/ -mapper /home/cloudera/tm.py -
reducer /home/cloudera/tr.py
17/12/10 14:20:17 WARN streaming.StreamJob: -jobconf option is deprecated, please use -D instead.
packageJobJar: [] [/usr/jars/hadoop-streaming-2.6.0-cdh5.4.2.jar] /tmp/streamjob3311542026245554702
.jar tmpDir=null
17/12/10 14:20:18 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
17/12/10 14:20:19 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
17/12/10 14:20:20 INFO mapred.FileInputFormat: Total input paths to process : 3
17/12/10 14:20:21 INFO mapreduce.JobSubmitter: number of splits:3
17/12/10 14:20:21 INFO Configuration.deprecation: mapred.reduce.tasks is deprecated. Instead, use m
apreduce.job.reduces
17/12/10 14:20:21 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1512848733301_0004
17/12/10 14:20:22 INFO impl.YarnClientImpl: Submitted application application_1512848733301_0004
17/12/10 14:20:22 INFO mapreduce.Job: The url to track the job: http://quickstart.cloudera:8088/pro
xy/application_1512848733301_0004/
17/12/10 14:20:22 INFO mapreduce.Job: Running job: job_1512848733301_0004
17/12/10 14:20:28 INFO mapreduce.Job: Job job_1512848733301_0004 running in uber mode : false
17/12/10 14:20:28 INFO mapreduce.Job:  map 0% reduce 0%
17/12/10 14:20:33 INFO mapreduce.Job:  map 33% reduce 0%
17/12/10 14:20:34 INFO mapreduce.Job:  map 67% reduce 0%
17/12/10 14:20:35 INFO mapreduce.Job:  map 100% reduce 0%
17/12/10 14:20:39 INFO mapreduce.Job:  map 100% reduce 100%
17/12/10 14:20:40 INFO mapreduce.Job: Job job_1512848733301_0004 completed successfully
17/12/10 14:20:40 INFO mapreduce.Job: Counters: 49
```

Рисунок 7 – Запуск с двумя reducerами

Представляет интерес исследование влияния количества mapper-ов и reducer-ов на скорость выполнения вычислений.

3. ЗАДАНИЕ НА РАБОТУ

3.1. Создать текстовые файлы в соответствии с вариантом задания. Варианты заданий приведены в таблице 1

3.2. Составить mapper в соответствии с вариантом задания.

3.2. Составить reducer.

3.3. Разместить написанные mapper и reducer по адресу /home/cloudera/ и сделать составленные файлы исполняемыми.

3.4. Запустить полученные mapper и reducer, запустить программу с большим числом reducerов.

3.6. Исследовать изменение параметров (затраченное время процессора, расход памяти) в зависимости от количества reducerов. Результаты исследования отразить в отчёте, сделать выводы

Таблица 1 – Варианты заданий

Вариант	Задание
1	Даны 4 файла с произвольным текстом, посчитать количество упоминаний каждого слова
2	Даны 3 файла с оценками для разных групп, структура данных: “Фамилия оценка”. Просуммировать количество оценок каждого балла.
3	Даны файлы с отзывами на фильм (в каждом файле более одного отзыва). Отзывы объемные, однако обязательно содержат слова “good” и “bad”. Подсчитать количество слов-оценок (“good” и “bad”) во всех файлах для составления рейтинга фильма.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Обоснуйте понятие MapReduce.

4.2. Какие шаги содержит алгоритм работы MapReduce?

4.3. Проиллюстрируйте принципы работы Map и Reduce?

4.4. Какие параметры следует учитывать при оценке производительности и эффективности использования Map и Reduce?

4.5.Перечислите преимущества и недостатки Hadoop MapReduce.

4.6. В чем состоит принцип «данные близко»? Как его использование влияет на архитектуру MapReduce?

4.7. Охарактеризуйте архитектуру «master-worker» и ее примерение относительно MapReduce.

4.8. Какие функции выполняет JobTracker?

4.9. Какие функции выполняет TaskTracker?

4.10. Опишите алгоритмы взаимодействия TaskTracker-узлов с узлом JobTracker и JobTracker-узла с клиентом (программным).

4.11. Что такое streaming-задача, как она связана с MapReduce?

4.12. Перечислите параметры запуска streaming-задачи и приведите пример создания задачи с аргументами.

ЛАБОРАТОРНАЯ РАБОТА №4
ИССЛЕДОВАНИЕ СПОСОБОВ РАБОТЫ С RDD И PAIR RDD В APACHE SPARK

1. ЦЕЛЬ РАБОТЫ

Исследовать возможности использования Apache Spark для обработки больших данных. Получить практические навыки работы с RDD и Pair RDD в Apache Spark.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Распределенные коллекции RDD

Для работы с данными в Spark используются распределенные коллекции элементов, Resilient Distributed Datasets (RDD). RDD во многом схожи с неизменяемыми коллекциями Scala и реализуют почти всех методы коллекций:

```
abstract class RDD[T] {
   def map[U](f: T => U): RDD[U] = ...
   def flatMap[U](f: T => TraversableOnce[U]): RDD[U] = ...
   def filter(f: T => Boolean): RDD[T] = ...
   def reduce(f: (T, T) => T): T = ...
}
```

2.2. Начало работы со Spark и создание RDD

Для работы с кластером в Spark используется экземпляр класса SparkContext, который можно создать следующим образом:

```
val sc: SparkContext = new SparkContext("local[*]", "spark_example")
```

Первый параметр – адрес кластера, к которому нужно присоединиться.

Значение "local" или "local[1]" означает, что будет использоваться локальный кластер с одним вычислительным блоком. Если указать "local[2]", будет использован локальный кластер с двумя вычислительными блоками. При значении "local[*]" будет создан кластер с количеством вычислительных блоков равным количеству логических ядер на компьютере. Для работы с локальными кластерами не нужно их отдельно создавать и настраивать, библиотека Spark берёт эту работу на себя.

Второй параметр – имя приложения, может быть любым.

2.3. Создание RDD

Создать новую коллекцию RDD можно двумя способами:

1. Преобразовать обычную коллекцию Scala в RDD:

```
val rdd1: RDD[Int] = sc.parallelize(List(1, 2, 3))
```

2. Считать данные из текстового файла:

```
val rdd2: RDD[String] = sc.textFile("/path/to/file")
```

В данном случае rdd2 является коллекцией всех строк в файле.

Также RDD можно получить из других RDD через преобразования: map, filter и т.д.

2.4. Примеры работы с RDD

Пусть есть коллекция строк – названий книг по программированию:

```
val books: RDD[String]
```

Узнать, сколько книг посвящены языку Scala можно следующим образом:

```
val result = books.filter(title => title.contains("Scala")).count()
```

Рассмотрим еще один пример. Пусть есть csv-файл вида:

1, Иванов, 22
2, Петров, 30
3, Сидоров, 65

4, Иваненко, 20
5, Петренко, 35
6, Сидоренко, 70

И есть класс, описывающий человека:
case class Person(name: String, age: Int)

Считать из файла коллекцию объектов класса Person можно следующим образом:

```
val rawPersons: RDD[String] = sc.textFile("/path/to/file")

val persons: RDD[Person] = rawPersons.map(strPerson => {
  strPerson.split(",").map(_.trim) match {
    case Array(_, name, age) => Person(name, age.toInt)
  }
})
println(persons.collect().mkString("\n"))
```

2.5. Преобразования и действия RDD

Все методы RDD делятся на две группы: преобразования (transformations) и действия (actions).

Преобразования в качестве результата возвращают новый RDD. Их вычисления ленивые (lazy) и не выполняются сразу. Это позволяет отсечь часть избыточных вычислений и не выполнять лишнюю работу.

Часто встречающиеся преобразования:

```
map[B](f: A => B): RDD[B]
flatMap[B](f: A => TraversableOnce[B]): RDD[B]
filter(predicate: A => Boolean): RDD[A]
distinct(): RDD[B]
groupBy[K](f: T => K): RDD[(K, Iterable[T])]
```

Преобразования над двумя RDD:

```
union(other: RDD[A]): RDD[A]  // Возвращает RDD, содержащий элементы обоих RDD
```

intersection(other: RDD[A]): RDD[A] // Возвращает RDD, содержащий элементы, которые присутствуют в обоих RDD
subtract(other: RDD[A]): RDD[A] // Возвращает исходный RDD, их которого удалены все элементы other
cartesian[B](other: RDD[B]): RDD[(A, B)] // Возвращает декартово произведение двух RDD

Действия производят операции над RDD и в качестве результата возвращают любой тип, кроме RDD. Они не ленивые (eager) и "запускают" вычисление над RDD, которые были подготовлены ранее вызванными преобразованиями.

Часто встречающиеся действия:
collect(): Array[A] // Возвращает все элементы RDD
count(): Long // Возвращает количество элементов в RDD
take(num: Int): Array[A] // Возвращает первые num элементов из RDD
reduce(op: (A, A) => A): A // Комбинирует элементы RDD друг с другом, используя функцию op и возвращает результат
foreach(f: A => Unit): Unit // Применяет функцию к каждому элементу RDD

Рассмотрим следующий пример кода:
```
val rdd = sc.textFile("/path/to/file")  // RDD с 1 000 000 записей
val temp = rdd.map(/*..*/).filter(/*..*/).map(/*..*/).distinct()
```

В результате его выполнения работа с данными будет только "запланирована", но еще не выполнена. Обработка всех 1 000 000 записей не произойдёт.

Если от промежуточного RDD temp вызвать метод take(10), то будут "запущены" ранее запланированные вычисления, но они затронут не всю коллекцию из 1 000 000 элементов, а только то количество исходных записей, достаточное для выполнения метода take(10).

Таким образом, "ленивая" природа преобразований над RDD позволяет отсечь лишние вычисления.

2.6. Кэширование вычислений в RDD

Из "ленивой" природы RDD вытекает особенность работы с ними. Вернемся к примеру со считыванием информации о людях из csv-файла и

созданию экземпляров класса Person:

```
val rawPersons: RDD[String] = sc.textFile("/path/to/file")

val persons: RDD[Person] = rawPersons.map(strPerson => {
  strPerson.split(",").map(_.trim) match {
    case Array(_, name, age) => Person(name, age.toInt)
  }
})
```

Допустим, мы хотим сформировать 3 "среза" коллекции persons в зависимости от возраста человека:

```
val young = persons.filter(_.age < 23).count()
val adult = persons.filter(23 to 55 contains _.age).count()
val old   = persons.filter(55 < _.age).count()
```

В данном случае для каждого среза нужно пройти цепочку rawPersons.map().filter().count(), то есть rawPersons.map() будет вычислен три раза. Хоть мы и сохранили результат rawPersons.map() в переменную persons, но метод map это "ленивое" преобразование и в переменной persons хранится только "план" будущих вычислений.

Spark позволяет закешировать часть вычислений, чтобы не выполнять лишнюю работу. Для этого нужно вызывать метод persist() после того преобразования, результаты которого хотим закешировать:

```
val persons: RDD[Person] = rawPersons.map(strPerson => {
  strPerson.split(",").map(_.trim) match {
    case Array(_, name, age) => Person(name, age.toInt)
  }
}).persist()

val young = persons.filter(_.age < 23).count()
val adult = persons.filter(23 to 55 contains _.age).count()
val old   = persons.filter(55 < _.age).count()
```

2.7. Операции редукции RDD

В Spark реализованы почти все операции редукции для RDD, которые

имеются в стандартных коллекциях Scala. Вот некоторые из них:

```
def fold(zeroValue: T)(op: (T, T) => T)
def reduce(f: (T, T) => T): T
def aggregate[U: ClassTag](zeroValue: U)(seqOp: (U, T) => U, combOp: (U, U)
=> U): U
```

Однако в Spark отсутствует реализация foldLeft и foldRight, т.к. эти операции нельзя распараллелить, а последовательная реализация будет неэффективной, потому что, из-за распределённой природы Spark, потребуется много синхронизаций данных между элементами кластера.

2.8. Коллекции Pair RDD

Ассоциативные массивы (англ. Map) – часто используемая структура для распределённой обработки данных. Например, на её использовании основывается технология MapReduce.

В Spark в качестве ассоциативных массивов выступает RDD[(K,V)], который в качестве значений содержит пары. Они называются Pair RDD.

Чтобы создать Pair RDD, достаточно преобразовать любой одиночный RDD, например с помощью метода map() или groupBy(). Для примера преобразуем RDD persons в Pair RDD, где ключом будет первая буква фамилии, а значением - объект Person:

```
val personsDictionary: RDD[(Char, Person)] = persons.map(
  person => (person.name.charAt(0), person)
)
println(personsDictionary.collect().mkString("\n"))
```

В консоль будет выведено следующее:

```
(И,Person(Иванов,22))
(П,Person(Петров,30))
(С,Person(Сидоров,65))
(И,Person(Иваненко,20))
(П,Person(Петренко,35))
(С,Person(Сидоренко,70))
```

2.8.1. Дополнительные методы для работы с Pair RDD

Для Pair RDD в Spark есть дополнительные методы, которые недоступны в обычном RDD. Вот некоторые из них:

```
def groupByKey(): RDD[(K, Iterable[V])]
def reduceByKey(func: (V, V) => V): RDD[(K, V)]
def mapValues[U](f: V => U): RDD[(K, U)]
def keys: RDD[K]
```

Все обозначенные выше методы являются преобразованиями. Для Pair RDD доступно особенное действие:

```
def countByKey(): Map[K, Long]
```

2.8.2. Виды join для Pair RDD

Для Pair RDD доступны еще несколько преобразований, объединяющие два Pair RDD по ключам:

```
def join[W](other: RDD[(K, W)]): RDD[(K, (V, W))]
def leftOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (V, Option[W]))]
def rightOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], W))]
def fullOuterJoin[W](other: RDD[(K, W)]): RDD[(K, (Option[V], Option[W]))]
```

Они отличаются только обработкой значений, чьих ключей нет в одном из объединяемых Pair RDD.

Ниже представлен пример применения различных видов join. Первая коллекция содержит в качестве ключа ID студента, а в качестве значения - фамилию студента. Вторая коллекция содержит в качестве ключа ID студента, а в качестве значения - предмет, по которому студент не аттестован. Для удобства чтения типов результирующих коллекций были объявлены псевдонимы типа String - Student и Subject

```
type Student = String
type Subject = String

val students: RDD[(Int, Student)] = sc.parallelize(List(
```

```
  (1, "Иванов"),
  (2, "Петров"),
  (3, "Сидоров")
))
val subjects: RDD[(Int, Subject)] = sc.parallelize(List(
  (1, "Программирование"),
  (1, "Физра"),
  (2, "Математика"),
  (4, "Физра")
))

val join: RDD[(Int,(Student, Subject))] = students join subjects
println(join.collect().sortBy(_._1).mkString("\n"))
  // (1,(Иванов,Программирование))
  // (1,(Иванов,Физра))
  // (2,(Петров,Математика))

val leftJoin: RDD[(Int,(Student, Option[Subject]))] = students leftOuterJoin
subjects
println(leftJoin.collect().sortBy(_._1).mkString("\n"))
  // (1,(Иванов,Some(Программирование)))
  // (1,(Иванов,Some(Физра)))
  // (2,(Петров,Some(Математика)))
  // (3,(Сидоров,None))

val rightJoin: RDD[(Int,(Option[Student], Subject))] = students rightOuterJoin
subjects
println(rightJoin.collect().sortBy(_._1).mkString("\n"))
  // (1,(Some(Иванов),Программирование))
  // (1,(Some(Иванов),Физра))
  // (2,(Some(Петров),Математика))
  // (4,(None,Физра))

val fullJoin: RDD[(Int,(Option[Student], Option[Subject]))] = students
fullOuterJoin subjects
println(fullJoin.collect().sortBy(_._1).mkString("\n"))
  // (1,(Some(Иванов),Some(Программирование)))
  // (1,(Some(Иванов),Some(Физра)))
```

```
// (2,(Some(Петров),Some(Математика)))
// (3,(Some(Сидоров),None))
// (4,(None,Some(Физра)))
```

3. ЗАДАНИЕ НА РАБОТУ

Внутри проекта в файлах **src/main/resources/person.csv** находится список людей. В каждой строке содержится запись о человеке в формате:

id_человека, имя_и_фамилия

В файле **src/main/resources/apartment.csv** находится список квартир. В каждой строке содержится запись о квартире в формате:

id_квартиры, id_человека, количество_комнат, адрес_квартиры

На каждого человека может быть оформлено 0 и более квартир. Если у человека нет оформленных на него квартир, то в файле apartment.csv нет записей с id этого человека.

Необходимо получить следующие коллекции:

1. Коллекция пар вида (количество_квартир, коллекция_id_людей у которых такое количество квартир). Коллекция должна быть упорядочена по количеству квартир по возрастанию.
2. Коллекцию пар вида (имя_и_фамилия, количество_квартир).
3. Коллекцию пар вида (адрес_квартиры, имя_и_фамилия). В коллекции должны содержаться адреса квартир с 2 и более комнатами.

После реализации требуемых вычислений необходимо исследовать возможности возможности RDD и PairRDD, реализовать и сравнить альтернативные способы решения задачи.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Что такое RDD?

4.2. Какие есть способы создания RDD?

4.3. В чем отличие между преобразованиями и действиями RDD?
4.4. Как выполняется кэширование вычислений в RDD?
4.5. Какие существуют операции редукции RDD?
4.6. Что такое Pair RDD?
4.7. Какие существуют разновидности операции join для Pair RDD?
4.8. Каким образом RDD представляются внутри Apache Spark?

ЛАБОРАТОРНАЯ РАБОТА №5
ИССЛЕДОВАНИЕ СПОСОБОВ ПЕРЕМЕЩЕНИЯ И ПАРТИЦИОНИРОВАНИЯ ДАННЫХ В APACHE SPARK

1. ЦЕЛЬ РАБОТЫ

Изучить особенности партицирования и перемещения данных при использовании RDD в Apache Spark. Исследовать влияние операций перемещения и партицирования на скорость выполнения операций выборки данных. Получить практические навыки применения партицирования в Apache Spark.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Перемещение данных

Перемещение данных – это процесс передачи данных по сети с одного узла кластера на другой. Перемещение происходит, например, при выполнении операции groupByKey, когда необходимо объединить данные с одинаковыми ключами в одну коллекцию и представить в виде пары типа (K, Iterable[V]). Изначально этих значений несколько и они могут быть распределены по всем узлам кластера, но, чтобы получить результирующую пару, необходимо собрать все эти значения на одном узле кластера.

Перемещение данных занимает относительно много времени и его нужно избегать.

В качестве примера обработаем данные о квартирах из предыдущей лабораторной работы. Создадим case-класс, представляющий информацию о количестве комнат в квартире человека:

```
case class ApartmentInfo(
  personId: Int,
  apartmentId: Int,
  roomsAmount: Int
)
```

Необходимо посчитать количество квартир и общее количество комнат для каждого человека.

Для начала решим задачу «в лоб». Для этого можно использовать

функцию groupByKey:

```
val apartmentStat = apartmentRdd.map(
  apartment => (apartment.personId, apartment.roomsAmount)
).groupByKey.map(
  pair => (pair._1, (pair._2.size, pair._2.sum))
).count()
```

На рисунке 1 изображена схема выполнения операции groupByKey.

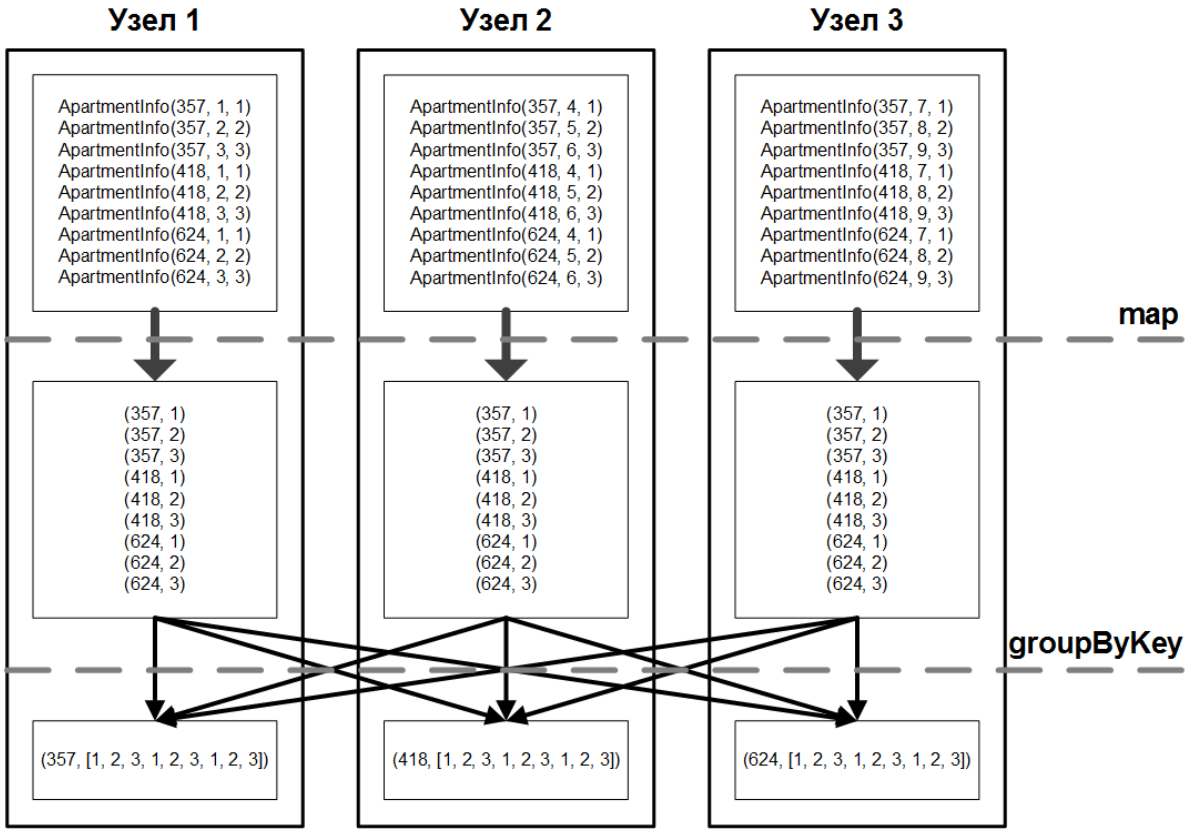


Рисунок 1 – Схема выполнения операции groupByKey

После выполнения операции map на каждом кластере образовались Pair RDD. Данные по каждому ключу расположены на всех узлах кластера, поэтому для выполнения операции groupByKey необходимо предварительно собрать все значения с одинаковым ключом на одном узле. На пересылку данных дополнительно расходуется значительное количество времени.

Попробуем уменьшить объем пересылаемых по сети данных. Для этого перепишем решение с использованием функции reduceByKey:

```
val apartmentStat2 = apartmentRdd.map(
  apartment => (apartment.personId, (1, apartment.roomsAmount))
).reduceByKey(
  (pair1, pair2) => (pair1._1 + pair2._1, pair1._2 + pair2._2)
).count()
```

На рисунке 2 изображена схема выполнения операции reduceByKey.

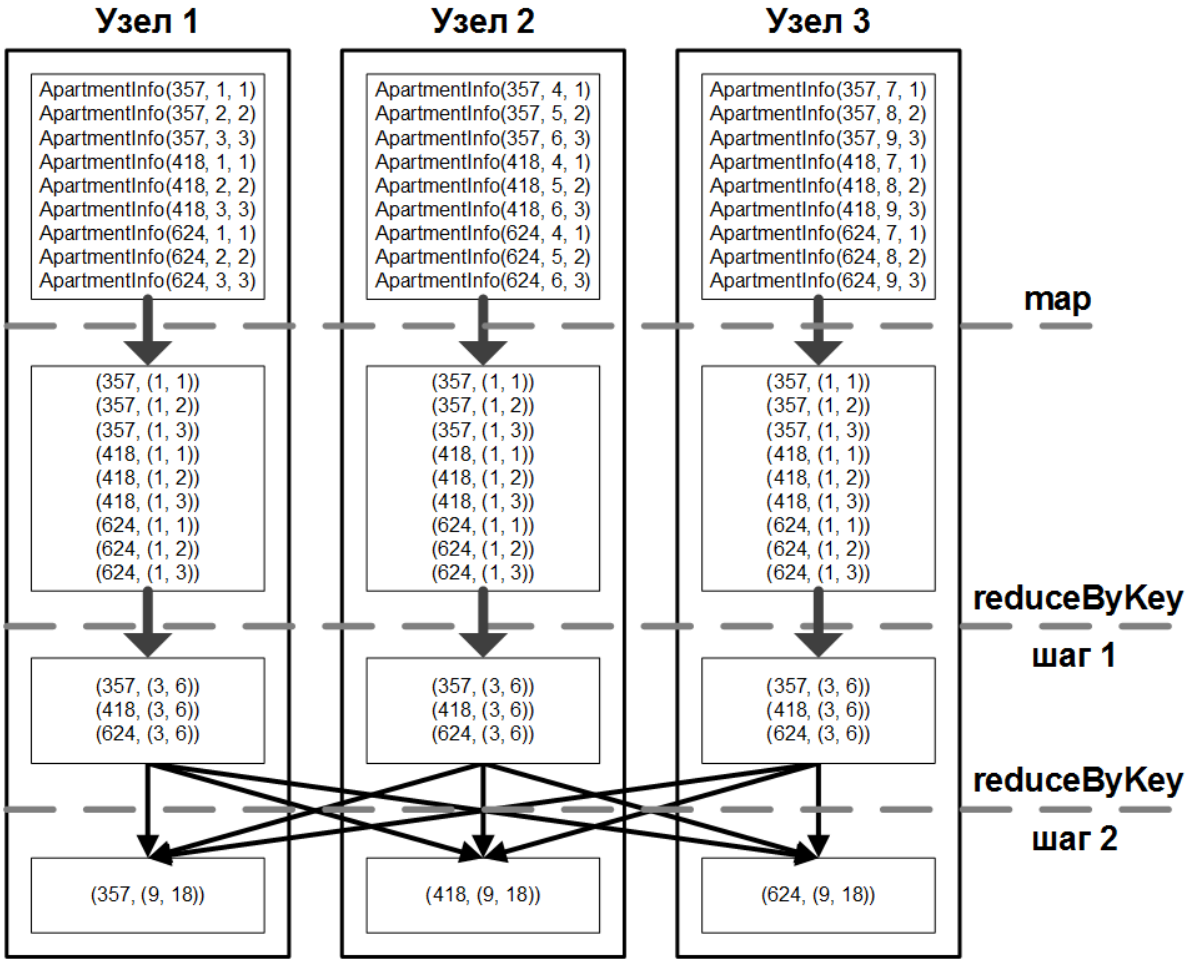


Рисунок 2 – Схема выполнения операции reduceByKey

Операция reduceByKey работает в два шага: сначала значения с одинаковым ключом «соединяются» локально, в пределах одного узла, затем происходит перемещение данных и окончательное объединений всех значений с одинаковым ключом. При этом по сети пересылается меньше данных, чем при использовании groupByKey.

В данной задаче реализация через reduceByKey оказалась быстрее в 2 раза. Тестирование проводилось локально на процессоре Intel(R) Core(TM)

i5-3210M 2.50GHz. Ниже представлен листинг кода, который спользовался для тестирования:

```
case class ApartmentInfo(
  personId: Int,
  apartmentId: Int,
  roomsAmount: Int
)

val apartmentRdd: RDD[ApartmentInfo] =
  sc.textFile(apartmentPath).map(strApartment => {
    strApartment.split(",").map(_.trim) match {
      case Array(personId, apartmentId, roomsAmount, _*) =>
        ApartmentInfo(
          personId.toInt,
          apartmentId.toInt,
          roomsAmount.toInt
        )
    }
  }).persist()

/* "прогрев" коллекции, чтобы компенсировать *
 * кеширование при последующих запусках      */
apartmentRdd.collect()

val apartmentStat1 = apartmentRdd.map(
  apartment => (apartment.personId, apartment.roomsAmount)
).groupByKey.map(
  pair => (pair._1, (pair._2.size, pair._2.sum))
)

val apartmentStat2 = apartmentRdd.map(
  apartment => (apartment.personId, (1, apartment.roomsAmount))
).reduceByKey(
  (pair1, pair2) => (pair1._1 + pair2._1, pair1._2 + pair2._2)
)

def time[R](block: => R): R = {
```

```
  val t0 = System.nanoTime()
  val result = block // call-by-name
  val t1 = System.nanoTime()
  val resultTime = BigDecimal((t1 - t0) / Math.pow(10, 9))
    .setScale(4, BigDecimal.RoundingMode.HALF_UP)
  println("Время: " + resultTime + " сек.")
  result
}

time(apartmentStat1.collect())  // Время: 0.6437 сек.
time(apartmentStat2.collect())  // Время: 0.3188 сек.
```

2.2. Партиционирование данных

Партиционирование – это распределение данных в Pair RDD по кластерам на основе ключей элементов. Получаемые в результате партиции обладают следующими свойствами:

1. Партиция хранится только на одном узле кластера. Если два элемента RDD находятся в одной партиции, то они гарантированно находятся на одном узле.
2. Каждый узел кластера содержит как минимум одну партицию.
3. Количество партиций для использования настраиваемо. По умолчанию оно равно общему количеству ядер на всех узлах кластера.

В Spark доступно два вида партиционирования:

1. По хешу (hash partitioning);
2. По интервалам (range partitioning).

Рассмотрим работу **партиционирования по хешу** на основе ранее написанного кода:

```
val apartmentStat1 = apartmentRdd.map(
  apartment => (apartment.personId, apartment.roomsAmount)
).groupByKey
```

Операция groupByKey работает в два этапа. Сначала вычисляет значение хеша для каждой пары (k, v). Это можно примерно описать следующим образом:

partition = k.hashCode() % partitionsCount

Затем все пары, принадлежащие одной и той же партиции, отправляются на узел, который содержит данную партицию.

Партиционирование по интервалам может применяться когда тип ключей поддерживает упорядоченность. Например, Int, Char, String. При этом будет учитываться отношение порядка между ключами, и заранее заданный набор интервалов значений ключей.

Рассмотрим оба способа партиционирования на примере. Допустим, есть Pair RDD с ключами [8, 96, 240, 400, 401, 800] и 4 партиции. Также, для упрощения, предположим, что функция хеширования это функция идентичности, то есть n.hashCode() == n.

При партиционировании по хешу элементы будут распределены по партициям следующим образом:

Партиция 0: [8, 96, 240, 400, 800]
Партиция 1: [401]
Партиция 2: []
Партиция 3: []

Как видно из примера, партиционирование по хешу может распределять элементы неравномерно, что скажется на производительности. Некоторые узлы кластера окажутся перегруженными, а некоторые будут простаивать.

Для рассмотрения партиционирования по интервалам предположим, что:

- все ключи неотрицательные,
- наибольший ключ в данных равен 800.

Тогда можно сформировать 4 интервала для партиционирования:

[1, 200], [201, 400], [401, 600], [601, 800].

Соответственно, элементы будут распределены следующим образом:
Партиция 0: [8, 96]
Партиция 1: [240, 400]

Партиция 2: [401]
Партиция 3: [800]

В результате элементы распределены более равномерно, чем при партиционировании по хешу.

2.2.1. Установка партиционирования для Pair RDD

Существует два способа задания партиционирования для Pair RDD:

1. Вызвать метод partitionBy и явно указать способ партиционирования в виде наследника класса Partitioner.
2. Использовать преобразование, которое вернёт Pair RDD с уже определённым способом партиционирования.

Рассмотим пример **явного партиционирования.** Допустим, есть Pair RDD следующего вида:

```
val personsRooms = apartmentRdd.map(
  apartment => (apartment.personId, apartment.roomsAmount)
)
```

Партиционировать данные можно следующим образом:

```
val partitioner = new RangePartitioner(4, personsRooms)
val partitioned = personsRooms.partitionBy(partitioner).persist()
```

Сначала мы создали партиционер, указав желаемое количество партиций (в данном случае 4). Также мы передали сами данные, которые будем партиционировать. Это необходимо для изучения ключей и формирование наиболее подходящих интервалов.

Также необходимо обратить внимание, что после получения партиционированной коллекции необходимо вызвать метод persist(), иначе коллекция будет постоянно партиционироваться при дальнейшей работе с ней.

Рассмотрим **преобразования, которые возвращают партиционированный RDD.** Некоторые преобразования возвращают RDD с уже установленным партиционированием. Например, sortByKey() использует RangePartitioner, a groupByKey() использует HashPartitoner.

Преобразования, которые сохраняют и передают дальше партиционирование:

- cogroup,

- groupWith,
- join,
- leftOuterJoin,
- rightOuterJoin,
- groupByKey,
- reduceByKey,
- foldByKey,
- combineByKey,
- partitionBy,
- sort,
- mapValues (если у родительского RDD был партиционер),
- flatMapValues (если у родительского RDD был партиционер),
- filter (если у родительского RDD был партиционер).

Все остальные преобразования вернут RDD без партиционирования, даже если коллекци ранее была партиционирована. Рассмотрим это на примере map():

```
pairRdd.map((k, v) => ("sample text", v)).
```

В данном случае, даже если бы map() передавал партиционирование дальше, оно все равно не имело бы смысла, т.к. поменялись ключи. Именно поэтому существует mapValues(), который позволяет делать map() над значениями, но не трогая ключи. Тем самым сохраняется партиционирование.

2.2.2. Оптимизация вычислений с помощью партиционирования

Использование партиционирования позволяет избежать лишних перемещений данных между узлами кластера, тем самым ускоряя процесс вычисления.

Рассмотрим вариант **комбинирования партиционирования и reduceByKey**. Добавим третий вариант реализации задания из раздела 2.1, использовав партиционирование. Ниже представлен дополненный листинг кода из раздела 2.1:

```
case class ApartmentInfo(
  personId: Int,
  apartmentId: Int,
  roomsAmount: Int
```

```
)

val apartmentRdd: RDD[ApartmentInfo] =
  sc.textFile(apartmentPath).map(strApartment => {
    strApartment.split(",").map(_.trim) match {
      case Array(personId, apartmentId, roomsAmount, _*) =>
        ApartmentInfo(
          personId.toInt,
          apartmentId.toInt,
          roomsAmount.toInt
        )
    }
  }).persist()

/* "прогрев" коллекции, чтобы компенсировать *
 * кеширование при последующих запусках      */
apartmentRdd.collect()

val apartmentStat1 = apartmentRdd.map(
  apartment => (apartment.personId, apartment.roomsAmount)
).groupByKey.map(
  pair => (pair._1, (pair._2.size, pair._2.sum))
)

val apartmentStat2 = apartmentRdd.map(
  apartment => (apartment.personId, (1, apartment.roomsAmount))
).reduceByKey(
  (pair1, pair2) => (pair1._1 + pair2._1, pair1._2 + pair2._2)
)

val personsRooms = apartmentRdd.map(
  apartment => (apartment.personId, apartment.roomsAmount)
)
val partitioner = new RangePartitioner(4, personsRooms)
val partitioned = personsRooms.partitionBy(partitioner).persist()
val apartmentStat3 = partitioned.mapValues(
  roomsAmount => (1, roomsAmount)
).reduceByKey(
```

```
  (pair1, pair2) => (pair1._1 + pair2._1, pair1._2 + pair2._2)
)

def time[R](block: => R): R = {
  val t0 = System.nanoTime()
  val result = block // call-by-name
  val t1 = System.nanoTime()
  val resultTime = BigDecimal((t1 - t0) / Math.pow(10, 9))
    .setScale(4, BigDecimal.RoundingMode.HALF_UP)
  println("Время: " + resultTime + " сек.")
  result
}

time(apartmentStat1.collect())  // Время: 0.6179 сек.
time(apartmentStat2.collect())  // Время: 0.3187 сек.
time(apartmentStat3.collect())  // Время: 0.2471 сек.
```

Вариант с применением партиционирования оказался быстрее аналогичной реализации без партиционирования. Стоит отметить две особенности реализации:

1. При создании RDD personsRooms в map() в качестве значения используется apartment.roomsAmount, а не (1, apartment.roomsAmount), как это изначально сделано в apartmentStat2. Такая реализация снижает количество передаваемых по сети данных при партиционировании. Перемещение коллекции типа RDD[(Int, Int)] требует меньше затрат чем перемещение коллекции типа RDD[(Int, (Int, Int))].

2. После партиционирование используется mapValues(), а не map(). Причины для этого рассмотрены в разделе 2.2.1.

Рассмотрим пример к**омбинирования партиционирования и join.** Допустим, есть музыкальный сервис, который хранит информацию о жанрах, на которые подписан пользователь. У него есть большая база подписок всех пользователей в виде коллекции пар (UserId, List[Genre]), где Genre - класс, представляющий музыкальный жанр:

```
userGenres: RDD[Int, List[Genre]] = sc.textFile(...).map(...).
```

Это приложение периодически получает информацию о композициях, которые прослушали пользователи, в виде небольшой (по сравнению с userGenres) коллекции, содержащей пары вида (UserId, MusicInfo), где

MusicInfo - класс с информацией о конкретной музыкальной композиции. На основании полученной информации необходимо посчитать сколько пользователей прослушали композиции не из своего списка жанров. Реализуем это в виде функции:

```
def readUpdates(updatesFileName: String) {
  val updates = sc.textFile(updatesFileName).map(...)
  val joined = userGenres.join(updates)

  joined.filter{
    case (userId, (genres, musicInfo)) =>
      !genres.contains(musicInfo.genre)
  }.count()
}
```

Данное решение неэффективно. Операция join ничего не знает о распределении ключей по узлам кластера. По умолчанию элементы обоих коллекций будут каждый раз распределяться по узлам кластера на основании хеша ключей, даже если «большая» коллекция не изменилась. Данная ситуация проиллюстрирована на рисунке 3.

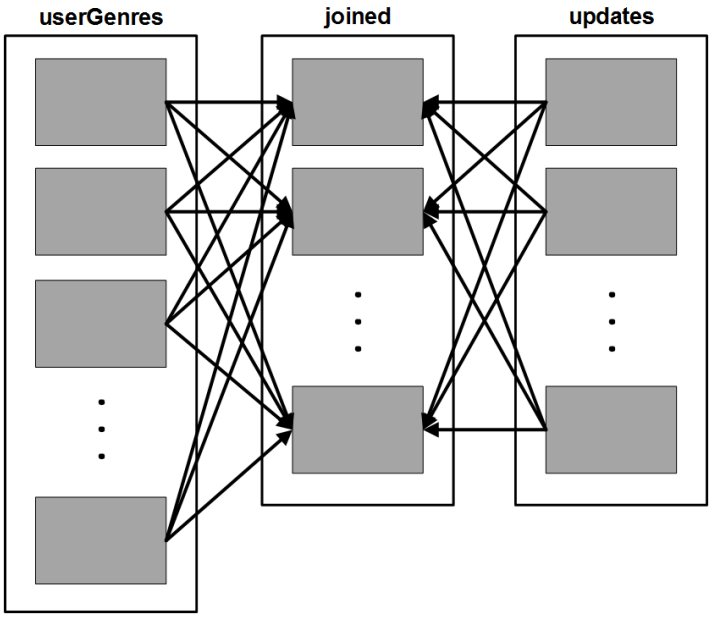


Рисунок 3 – Неэффективное объединение двух коллекций

Чтобы избавиться от лишнего перемещения данных, нужно предварительно применить партиционирование к userGenres. В таком случае элементы коллекции будут распределены по узлам кластера один раз, а при каждом вызове readUpdates по сети будут перемещаться только элементы updates.

Визуально это изображено на рисунке 4.

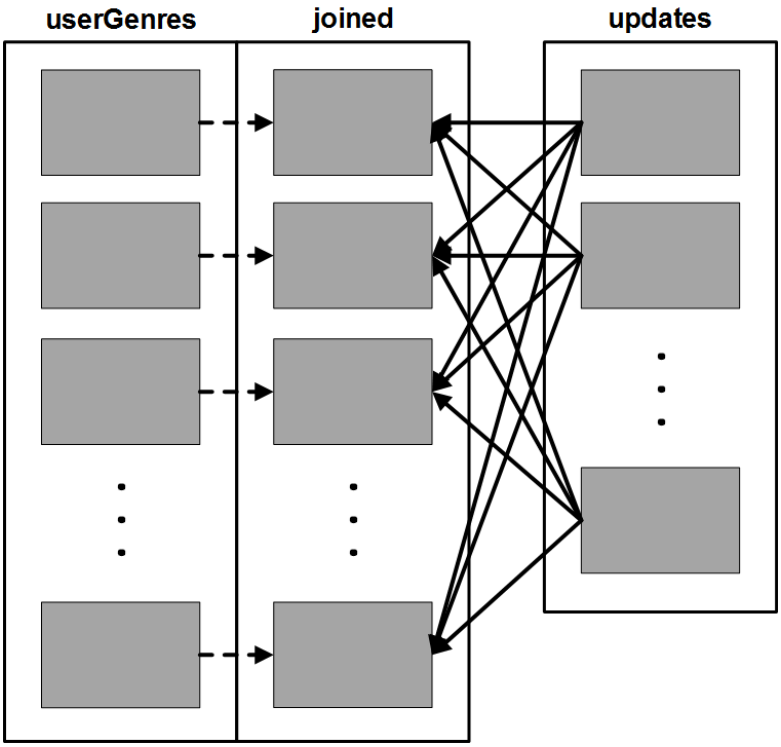


Рисунок 4 – Объединение двух коллекций с предварительным партиционированием

Пунктирные стрелки на рисунке 4 означают логическое «перемещение» данных. На самом деле данные уже распределены по узлам кластера и перемещения не происходит.

Более того, если взять две гипотетические коллекции и партиционировать их одним и тем же партиционером, то в дальнейшем все операции join между ними будут происходить локально в пределах узла кластера, без перемещения данных.

Для примера с музыкальным сервисом партиционирование обеих коллекций не подходит, т.к. вторая коллекция для объединения каждый раз новая, и предварительное её партиционирование не принесёт выигрыша в производительности.

2.2.3. Операции, которые могут повлечь перемещение данных

Рассмотрим **операции, которые могут повлечь перемещение данных.** Перемещение данных происходит, если результирующая коллекция зависит от других элементов из этой же или другой коллекции. Однако, с помощью правильного применения партиционирования можно избежать

перемещения данных. Поэтому про некоторые операции говорят, что они только *могут* повлечь перемещение данных.К таким операциям относятся:

- cogroup,
- groupWith,
- join,
- leftOuterJoin,
- rightOuterJoin,
- groupByKey,
- reduceByKey,
- combineByKey,
- distinct,
- intersection,
- repartition,
- coalesce.

Отсюда следует, что организация данных в кластере оказывает сильное влияние на скорость вычислений.

2.3. Граф вычислений. Широкие и узкие зависимости

Внутри Spark обработка RDD и вычисление конечных результатов представляются в виде графа вычислений. Например, для данного фрагмента кода граф вычислений изображён на рисунке 5:

```
val apartmentRdd: RDD[ApartmentInfo] =
  sc.textFile(apartmentPath).map(...).persist()

val apartmentStat1 = apartmentRdd.map(
  apartment => (apartment.personId, apartment.roomsAmount)
).groupByKey.map(
  pair => (pair._1, (pair._2.size, pair._2.sum))
)

val apartmentStat2 = apartmentRdd.map(
  apartment => (apartment.personId, (1, apartment.roomsAmount))
).reduceByKey(
  (pair1, pair2) => (pair1._1 + pair2._1, pair1._2 + pair2._2)
)
```

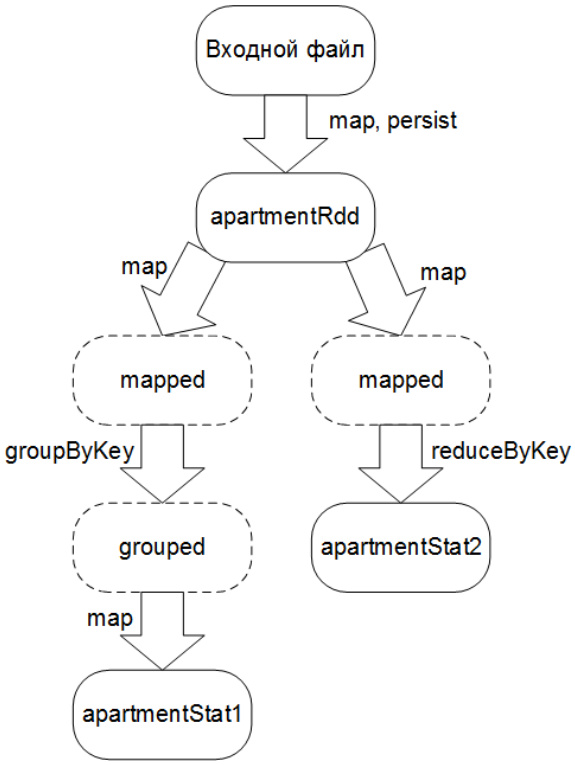


Рисунок 5 – Пример графа вычислений

Представление RDD включает в себя 4 аспекта:

- партиции – блоки данных,

- зависимости – описывают связь текущего RDD и его партиций с RDD, от которого он был получен,

- функция для перехода от родительского RDD,

- метаданные о схеме партицирования и распределении данных по узлам кластера.

Преобразования над RDD вызывают являются причиной перемещения данных между узлами кластера. Преобразования делятся на два вида по **типу зависимости**:

1. Узкая зависимость - каждая партиция родительского RDD **используется максимум одной** партицией дочернего RDD.

2. Широкая зависимость - каждая партиция родительского RDD **может использоваться несколькими** партициями дочернего RDD.

На рисунке 6 изображены примеры типов зависимостей.

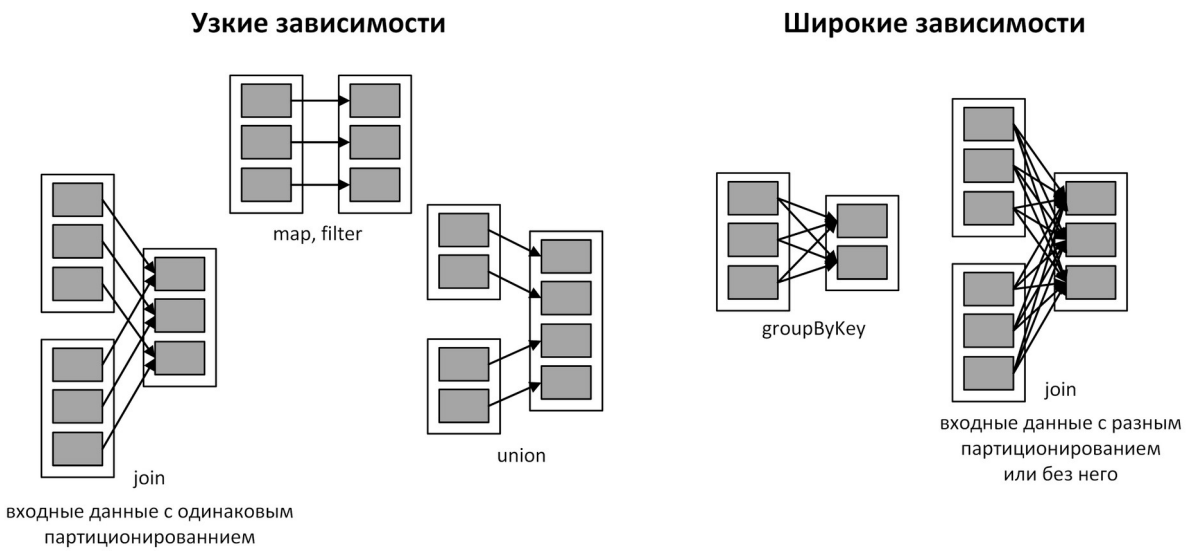


Рисунок 6 – Примеры типов зависимостей

Преобразования с узкими зависимостями выполняются быстро и не требуют перемещения данных по сети. Преобразования с широкими зависимостями выполняются медленно и требуют перемещения по сети либо всего объёма данных, либо какой-то его части.

К преобразованиям с узкими зависимостями относятся:

- map,
- mapValues,
- flatMap,
- filter,
- mapPartitions,
- mapPartitionsWithIndex.

Преобразования из пункта 2.2.3 в ситуациях, когда осуществляется перемещение данных, являются преобразованиями с широкими зависимостями.

Просмотреть граф зависимостей конкретной коллекции можно с помощью метода toDebugString.

2.3.1. Отказоустойчивость с применением графа вычислений

На рисунке 7 представлен более подробный пример графа вычислений гипотетической программы. Стоит отметить, что после выполнения groupBy коллекция B по сути содержит внутри себя уже партиционированные данные. В связи с этим логично вызвать persist на B и, тем самым, «сохранить» партиционирование. Это объясняет почему в операции join между B и F зависимость от B является узкой зависимостью, а зависимость

от F – широкой.

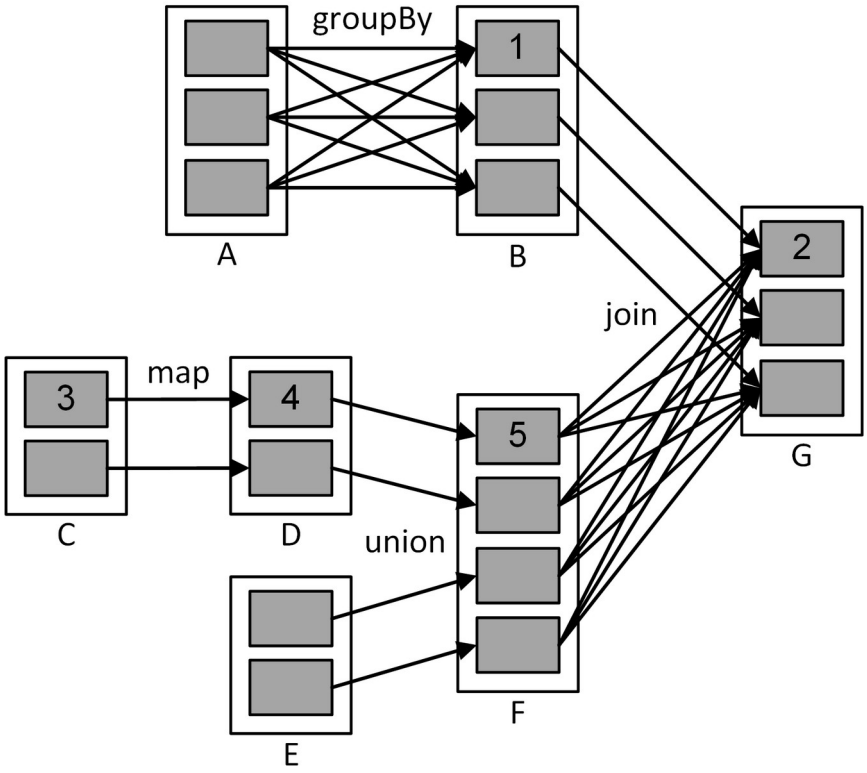


Рисунок 7 – Более подробный граф вычислений

Отказоустойчивость в Spark основывается на особенностях функционального программирования и графах вычислений.

Используемые свойства функционального программирования:

1. Используемые коллекции (RDD) являются неизменяемым (immutable) типом данных.

2. В работе с коллекциями применяются функции высшего порядка для функционального преобразования неизменяемых данных.

3. Функции для вычисления новой коллекции основываются на родительской коллекции и являются частью представления коллекции внутри Spark.

Граф вычислений, помимо всего прочего, содержит информацию о зависимостях между партициями разных коллекций.

Информация о зависимостях между партициями и обозначенные выше три особенности функционального программирования позволяют организовать отказоустойчивость программы путём повторного вычисления потерянных при сбое партиций.

Например, представим, что партиция 5 вышла из строя. В таком случае, потребуется пересчитать только элементы партиции 3 и 4, а не полностью все

коллекции C, D, E и F. Пересчитывание ошибочных партиций происходит очень быстро для узких зависимостей.

Однако, для широких зависимостей пересчитывание ошибочных партиций не эффективно. Рассмотрим на примере партиции 2: если она вышла из строя, то для ее восстановление понадобится полностью пересчитать коллекции C, D, E и F. Коллекция B закеширована, поэтому пересчитывать ее партицию 1 ненужно.

3. ЗАДАНИЕ НА РАБОТУ

3.1. Замерить время работы программ из предыдущей лабораторной работы.

3.2. Добавить партиционирование и, если возможно, заменить операции с лишним перемещением данных на аналоги без лишнего перемещения данных.

3.3. Замерить новое время работы программы после внесения изменений. Исследовать влияние партицирования и перемещения данных на скорость работы программы.

3.4. Нарисовать граф вычислений до и после внесённых изменений.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Что понимается под операцией перемещения данных в Apache Spark?

4.2. Приведите пример использования операций groupByKey и reduceByKey.

4.3. В чем заключается операция партицирования данных в Apache Spark?

4.4. Какие существуют виды партицирования данных в Apache Spark?

4.5. Каким образом задается партицирование для Pair RDD?

4.6. Каким образом партицирование позволяет оптимизировать вычисления?

4.7. Какие операции могут повлечь перемещение данных?

4.8. Как обеспечивается отказоустойчивость в Apache Spark?

ЛАБОРАТОРНАЯ РАБОТА №6
ИССЛЕДОВАНИЕ СПОСОБОВ ИСПОЛЬЗОВАНИЯ ОБЛАЧНЫХ КОНТЕЙНЕРОВ

1. ЦЕЛЬ РАБОТЫ

Получить практические навыки использования облачных контейнеров на примере Docker. Исследовать возможности Docker для контейнеризации веб-приложений.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

Docker – проект с открытым исходным кодом для автоматизации разработки, развертывания и эксплуатации приложений. Docker обеспечивает истинную независимость между приложениями и инфраструктурой, а также разработчиками и ИТ-подразделениями. Используя Docker для доставки, тестирования и развертывания кода, можно значительно сократить задержку между написанием кода и запуском его в продакшене.

С помощью Docker возможно упаковывать и запускать приложения в свободно изолированной среде – **контейнере**. Изоляция и безопасность позволяют запускать множество контейнеров одновременно на данном хосте. Легковесность контейнера позволяет запускать больше контейнеров, по сравнению с виртуальной машиной.

Платформа и средства контейнерной виртуализации могут быть полезны в следующих случаях:

1. Быстрое выкладывание приложений. Docker упрощает жизненный цикл разработки, позволяя разработчикам работать в стандартизированных средах с использованием локальных контейнеров, которые предоставляют приложения и службы. Контейнеры отлично подходят для непрерывной интеграции и непрерывной работы (CI / CD).

2. Отзывчивое развертывание и масштабирование. Контейнерная платформа Docker позволяет работать с высокой переносимостью. Контейнеры-докеры могут работать на локальном ноутбуке разработчика, на физических или виртуальных машинах в дата-центре, на облачных

провайдерах или в смешанных средах.

Портативность порталов и легкий вес также позволяют легко динамически управлять рабочими нагрузками, масштабировать или отключать приложения и услуги, как того требуют бизнес-логика, в режиме реального времени.

3. Выполнение большего количества рабочих нагрузок на одном и том же оборудовании. Docker – легкий и быстрый. Он предоставляет устойчивую, рентабельную альтернативу виртуальным машинам на основе гипервизора. Docker идеально подходит как для сред с высокой нагрузкой, так и маленький и средних приложений.

2.1. Архитектура Docker

Docker использует архитектуру клиент-сервер. Docker-клиент общается с Docker-демоном, который отвечает за создание, запуск и распространение Docker-контейнеров. Клиент и демон Docker могут работать в одной системе, или можно подключить Docker-клиент к удаленному Docker-демону. Клиент и демон Docker обмениваются данными с помощью REST API, сокетов UNIX или сетевого интерфейса.

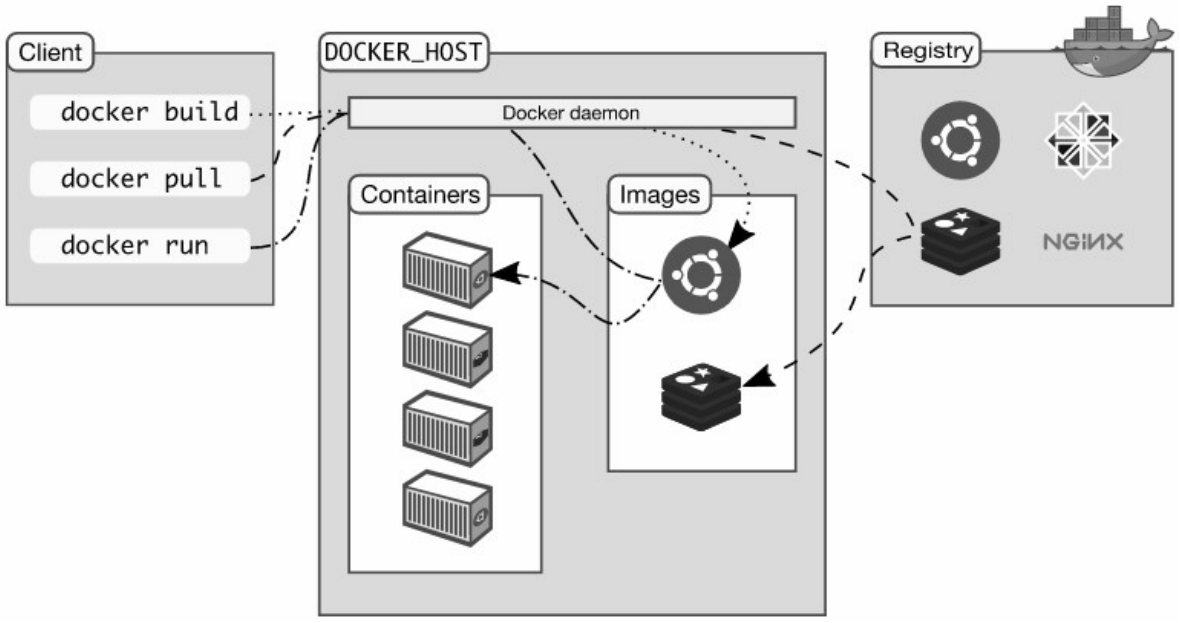


Рисунок 1 – Архитектура Docker

Docker-демон (dockerd) прослушивает запросы Docker API и управляет объектами Docker, такими как изображения, контейнеры, сети и тома. Демон

может также взаимодействовать с другими демонами для управления службами Docker.

Docker-клиент (docker) является основным способом взаимодействия многих пользователей с Docker. При запуска docker, клиент отправляет эти команды в dockerd, который их выполняет. Команда docker использует Docker API. Docker-клиент может общаться с несколькими демонами.

В **Docker-реестре** хранятся Docker-образы. Docker Hub (https://hub.docker.com/) и Docker Cloud (https://cloud.docker.com/) – это публичные реестры, на которых хранится огромная коллекция готовых Docker-образов. При запуске докеров, требуемые образы вытягиваются из настроенного реестра, а используя команду push docker, образ переводится в настроенный реестр.

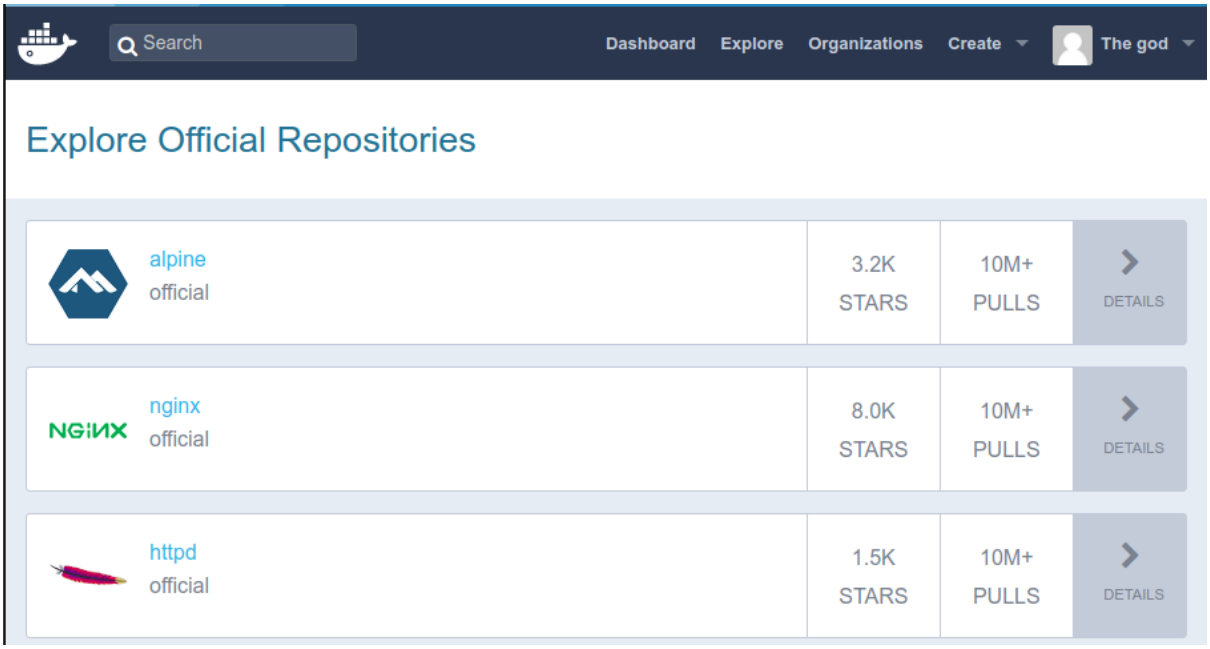


Рисунок 2 – Список официальных образов на Docker Hub

Docker-образ представляет собой read-only шаблон с инструкциями по созданию Docker-контейнера . Часто образ основан на другом образе с дополнительными параметрами. Например, можно создать образ, основанный на образе ubuntu, но устанавливающий веб-сервер Apache.

Для создания собственного образа необходимо создать файл Docker с простым синтаксисом для определения шагов, необходимых для создания образа и его запуска. Каждая команда в файле Docker создает слой в образе.

Контейнер представляет собой исполняемый экземпляр **Docker-образа**, в нем содержится все, что необходимо для работы приложения. Контейнеры можно создавать, запускать, останавливать, перемещать или

удалять. Каждый контейнер относительно хорошо изолирован от других контейнеров и его хост-машины. Контейнер определяется его образом, а также параметрами конфигурации, указанными при создании или запуске. При удалении контейнера любые изменения в его состоянии, которые не хранятся в постоянном хранилище, исчезают.

2.2. Установка Docker

Docker доступен для скачивания в двух версиях Community Edition (CE) и Enterprise Edition (EE).

Docker Community Edition (CE) идеально подходит разработчикам и небольшим группам, которые хотят начать работу с Docker и экспериментировать с приложениями на базе контейнеров.

Docker Enterprise Edition (EE) предназначен для разработчиков и ИТ-специалистов, которые строят, обрабатывают и запускают критически важные приложения для бизнеса в масштабе производства.

Подробную информацию о системных требованиях, поддерживаемых платформах можно найти в официальной документации (https://docs.docker.com/install/).

Рассмотрим пример установки Docker CE на Ubuntu 16.04 LTS. Для этого необходимо последовательно выполнить следующие команды:

```
sudo apt-get remove docker docker-engine docker.io
sudo apt-get update
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"
sudo apt-get update
sudo apt-get install docker-ce
```

Чтобы убедиться, что Docker CE установлен правильно, можно

запустить образ hello-world, как показано на рисунке 3.

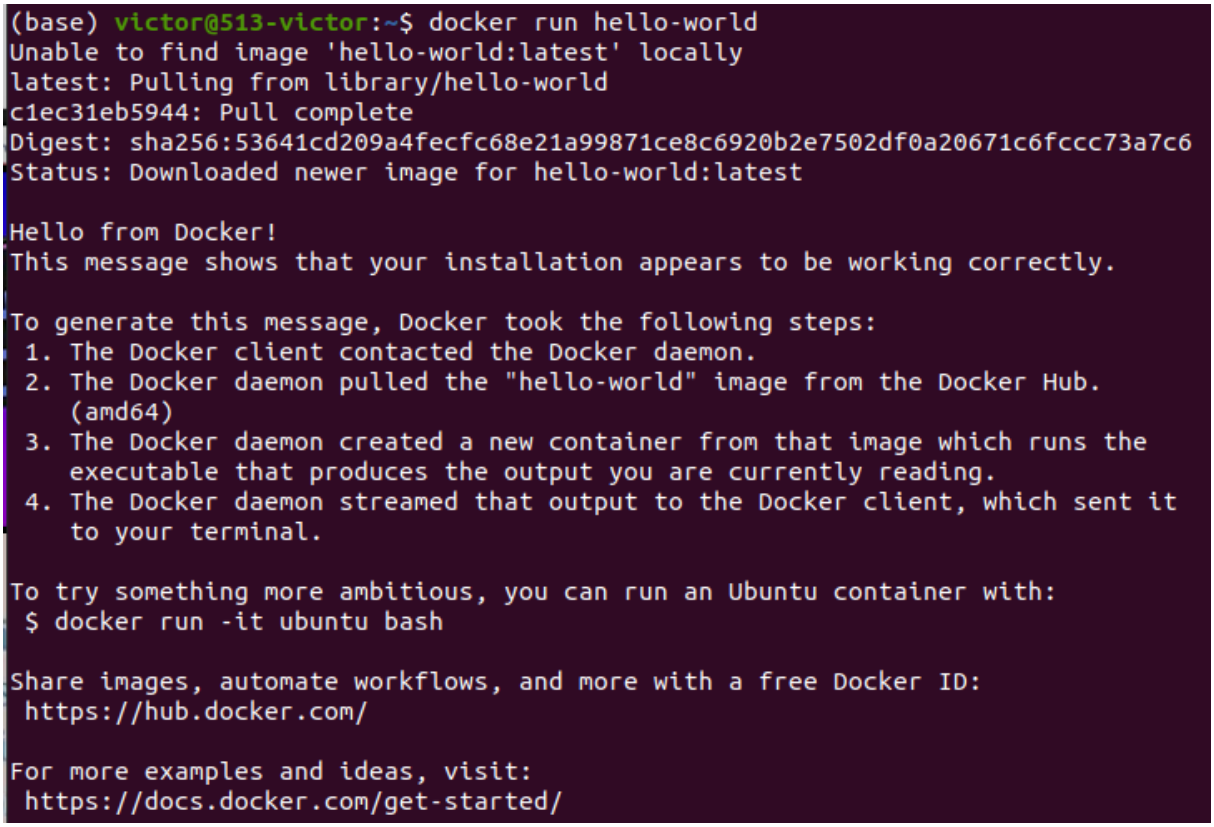


Рисунок 3 – Запуск Docker-образа hello-world

Для использования команд Docker без прав sudo можно ввести следующую команду: sudo chmod ug+s /usr/bin/docker

2.3. Основные команды Docker

В таблице 1 приведен список основных команд для работы с Docker-контейнерами. Список команд для работы с Docker-образами приведен в таблице 2.

Таблица 1 – Основные команды для работы с Docker-контейнерами

Команда	Описание
docker run --name [container-id] -d -p 80:80 [image name]	Запуск контейнера с образа на 80 порту
docker ps -a	Просмотр запущенных и остановленных контейнеров

Продолжение таблицы 1

Команда	Описание
docker attach [container-id]	Для подсоединения к запущенному контейнеру
docker start [container-id]	Запуск контейнера
docker stop [container-id]	Остановка запущенного контейнера
docker restart [container-id]	Перезагрузка запущенного контейнера
docker kill [container-id]	Выключение запущенного контейнера
docker rm [container-id]	Удаление остановленного контейнера
docker rm `docker ps -a -q`	Удалить все оставшиеся контейнеры
docker commit [container-id] [image name]	Создание нового образа с контейнера
docker cp [container-id]:/etc/passwd .	Копирование файла с контейнера на хост
docker inspect [container-id]	Просмотр информации о контейнере
docker inspect --format="{{.NetworkSettings.IPAddress}}" [container-id]	Посмотреть ip адрес запущенного контейнера
docker logs [container-id]	Посмотреть log контейнера
docker top [container-id]	Посмотреть список процессов контейнера
docker rename [default name] [new name]	Переименовать контейнер
docker history [container-id]	История команд для данного контейнера

Таблица 2 – Основные команды для работы с Docker-образами

Команда	Описание
docker build -t [image name] .	создание образа с помощью build файла.
docker search <WORD>	Поиск образа по ключевому слову
docker images	Список всех локальных образов
docker commit [container-id] [new-image-name]	Запись измененного контейнера под другим именем в локальное хранилище
docker pull [image name]	Скачать последний образ в локальное хранилище
docker pull [image name][:number]	Скачать образ версии NUMBER в локальное хранилище
docker rmi [image-id]	Удалить контейнер с локального хранилища
docker rmi -f [image-id]	Удалить запущенный контейнер с локального хранилища
docker rmi $(docker images -q)	Удалить ВСЕ образы
docker load < /tmp/myimage.tar	Создание образа с tar архива с STDIN
docker save [image name] > /tmp/myimage.tar	Запись образа в tar архив в STDOUT
docker import http://example.com/exampleimage.tgz	Импорт образа с удаленного файла
docker import - exampleimagelocal:new	Импорт с локального файла
docker import - exampleimagedir	Импорт с локальной директории

2.4. Использование Docker

2.4.1. Запуск контейнера в Docker

При установке уже был рассмотрен процесс запуска контейнера hello-

world. Рассмотрим процесс запуска контейнера и основных команд для этого более детально.

Для примера запустим новый контейнер:

docker run ubuntu /bin/echo 'I`am learning Docker!'

Здесь:

docker run – это команда запуска контейнера;

ubuntu – образ, который запускается (образ операционной системы Ubuntu);

/bin/echo 'I`am learning Docker!' – команда, которая будет запускаться внутри нового контейнера. Данный контейнер просто выводит 'I`am learning Docker!' и останавливает выполнение.

Результат выполнения показаны на рисунке 4.

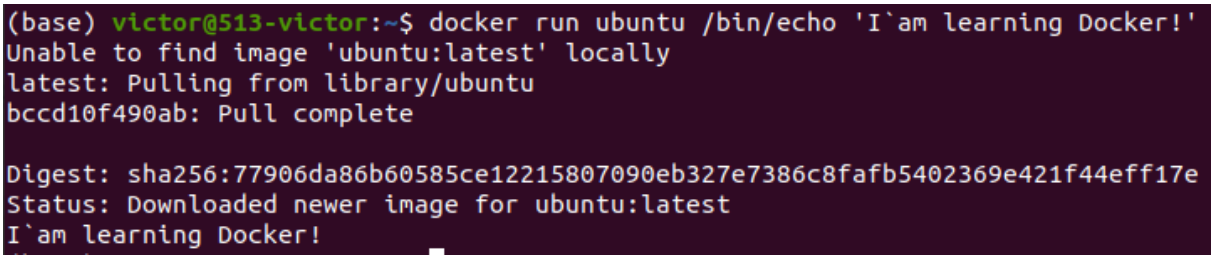


Рисунок 4 – Запуск контейнера

Если необходимо, чтобы контейнер работал после окончания сеанса, необходимо «демонизировать» его, для этого выполним команду:

docker run --name daemon -d ubuntu /bin/sh -c "while true; do echo Still running; sleep 1; done"

Здесь:

--name daemon назначает имя новому контейнеру. Если его не указать, имя генерируется и назначается автоматически;

-d запускает контейнер в фоновом режиме («демонизирует» его).

Для просмотра текущих контейнеров выполним команду:

docker ps -a

Результат выполнения приведен на рисунке 5.

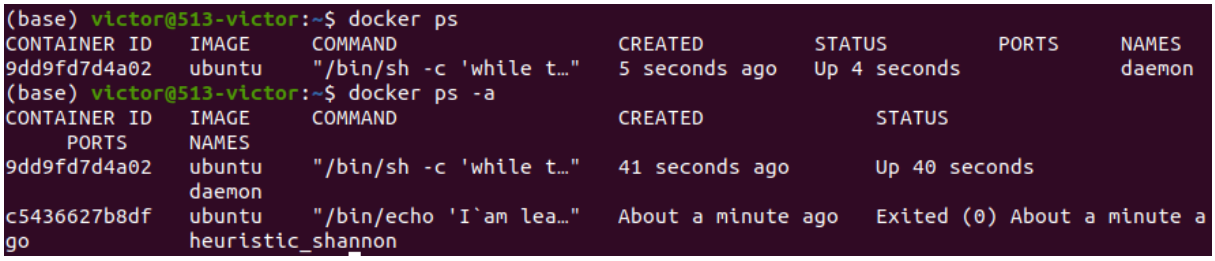


Рисунок 5 – Список существующих контейнеров

Здесь:

docker ps – команда для перечисления контейнеров;

-a показывает все контейнеры (без -a ps покажет только запущенные контейнеры).

Для просмотра результатов работы запущенного контейнера-демона выполним команду:

docker logs -f daemon

Результаты выполнения команды показаны на рисунке 6.

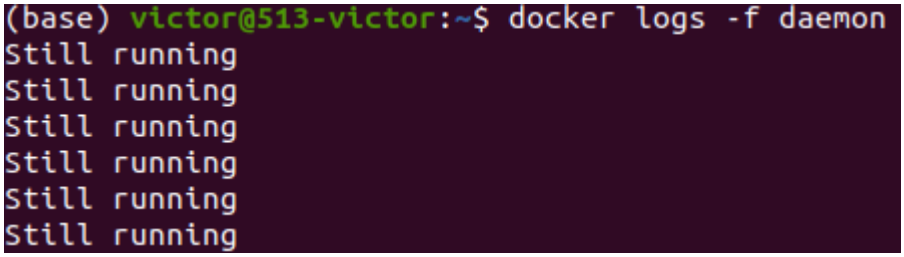


Рисунок 6 – Результат работы запущенного контейнера-демона

Для остановки и удаления контейнера-демона воспользуемся командами:

docker stop daemon

docker rm daemon

Для удаления всех контейнеров:

docker rm $(docker ps -a -q)

Здесь:

docker rm – удаление контейнера;

-f (для rm) остановка контейнер, если он работает (принудительное удаление);

-q (для ps) – это вывод только идентификаторов контейнера.

2.4.2. Создание образа в Docker

Создадим Docker-образ с помощью Dockerfile. Он содержит набор инструкций, чтобы сообщить Docker, как создавать и настраивать контейнер. Создадим каталог с именем docker, внутри этого каталога создаем файл Dockerfile без расширения и добавим в него следующие команды:

FROM ubuntu:latest

Данная команда говорит Docker, что мы собираемся создать новый

образ, используя ubuntu:latest в качестве базового изображения.

Обновим apt-get и использовать его для установки nodejs и npm:

```
RUN apt-get update
RUN apt-get install -y nodejs
RUN apt-get install -y npm
```

http-server – это простой HTTP-сервер командной строки с нулевой конфигурацией. Установим его с помощью npm:

```
RUN npm install -g http-server
```

Создадим символическую ссылку в каталог nodejs, чтобы ее можно было запустить с помощью команды node:

```
RUN ln -s /usr/bin/nodejs /usr/bin/node
```

Создаем index.html с некоторым содержимым в нашем каталоге и добавим этот файл в контейнер:

```
ADD index.html /usr/apps/myApp/index.html
```

Устанавливаем рабочий каталог контейнера в этот путь:

```
WORKDIR /usr/apps/myApp/
```

Указываем команду для запуска http-сервера в бесшумном режиме, указав флаг -s:

```
CMD ["http-server", "-s"]
```

Итоговый Dockerfile должен выглядеть следующим образом:

```
FROM ubuntu:latest
RUN apt-get update
RUN apt-get install -y nodejs
RUN apt-get install -y npm
RUN ln -s /usr/bin/nodejs /usr/bin/node
RUN npm install -g http-server
ADD index.html /usr/apps/myApp/index.html
WORKDIR /usr/apps/myApp/
CMD ["http-server", "-s"]
```

После этого для сборки Docker-образа выполняем следующую команду

в каталоге:
docker build -t bdot_lr6:docker_page .

Эта команда создает образ контейнера в bdot_lr6 репозитории и помечает его тегом docker_page. Для запуска контейнера необходимо выполнить следующую команду:
docker run -t -p 8008:8080 bdot_lr6:docker_page

Если все сделано правильно и запуск прошел успешно, то по адресу 0.0.0.0:8080 можно увидеть содержимое файла index.html, как показано на рисунке 7.

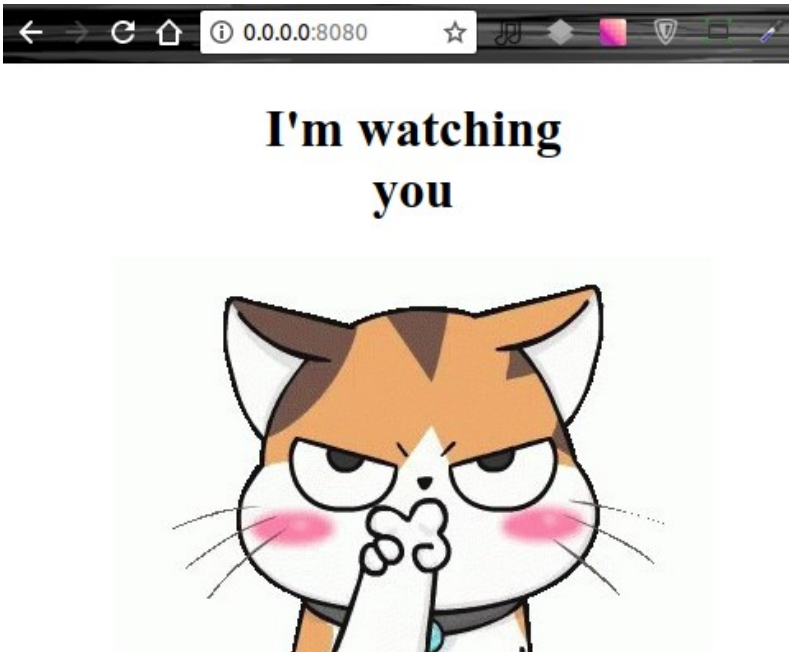


Рисунок 7 – Результат работы контейнера

3. ЗАДАНИЕ НА РАБОТУ

3.1. С помощью Dockerfile или docker-compose создать образ для развертывания статической html-страницы (может содержать nginx, apache, nodejs).

3.2. Содержание html-страницы выбирается произвольно. По желанию можно добавить стили оформления.

3.3. С помощью команды build создать контейнер из написанного образа. Вывести на экран список существующих образов и запущенных контейнеров с помощью команд docker ps, docker images. Исследовать

возможности управления Docker-контейнером.

3.4. Зарегистрировать аккаунт на DockerHub (https://hub.docker.com/) и загрузить туда полученный контейнер.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

4.1. Что такое Docker ?
4.2. Для чего применяется Docker?
4.3. Перечислите основные компоненты Docker и их назначение.
4.4. Для чего используется DockerHub?
4.5. В чем отличие Docker от виртуальной машины?
4.6. Преимущества и недостатки Docker?
4.7 Для чего служит Docker Compose?
4.8. Перечислите основные команды, исползуемые в Dockerfile.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Apache Hive [Электронный ресурс]: Official development web site / Дата обращения: 21.12.2017. — Режим доступа: http://hive.apache.org/

2. Cloudera QuickStart VMs [Электронный ресурс]: Official web site / Дата обращения: 10.02.2018. — Режим доступа: https://www.cloudera.com/downloads/quickstart_vms/5-12.html

3. Хабрахабр [Электронный ресурс]: Дата обращения: 02.03.2018 — Режим доступа: https://habrahabr.ru/post/74792/

4. IBM [Электронный ресурс]:Official web site Дата обращения: 02.03.2018 — Режим доступа: https://www.ibm.com/developerworks/ru/library/bd-mapreduce/index.html

5. Coursera [Электронный ресурс]: Official web site/ Дата обращения 12.02.2018. - Режим доступа: https://www.coursera.org/learn/hadoop

6. Semantikoz [Электронный ресурс]: The Free Hive Book / Дата обращения: 21.12.2017. — Режим доступа: http://www.semantikoz.com/blog/the-free-apache-hive-book/

7. Apache [Электронный ресурс]: Apache Hive Wiki / Дата обращения: 21.12.2017.—Режим доступа: https://cwiki.apache.org/confluence/display/Hive/Home

8. Taskdata [Электронный ресурс]: Official web site / Дата обращения: 12.02.2018. — Режим доступа: http://www.taskdata.com/index.php?id=26&Itemid=5&option=com_content&view=article&lang=ru

9. The apache software foundation [Электронный ресурс]: HDFS Architecture Guide / Дата обращения: 21.12.2017.—Режим доступа: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

10. The apache software foundation [Электронный ресурс]: HDFS Commands Guide / Дата обращения: 21.12.2017.—Режим доступа: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSCommands.html

11. The apache software foundation [Электронный ресурс]: HDFS Java API / Дата обращения: 21.12.2017.—Режим доступа: http://hadoop.apache.org/core/docs/current/api/

12. Install Docker [Электронный ресурс]: Docker Documentation / Дата обращения: 22.02.2018. — Режим доступа: https://docs.docker.com/install/

13. Docker Get Started [Электронный ресурс]: Docker Documentation / Дата обращения: 22.02.2018. — Режим доступа: https://docs.docker.com/get-

started/

14. Docker Hub [Электронный ресурс]: Official web site / Дата обращения: 22.02.2018. — Режим доступа: https://hub.docker.com/

15. Docker Cloud [Электронный ресурс]: Official web site / Дата обращения: 22.02.2018. — Режим доступа: https://cloud.docker.com/

16. Григорьев, А. А. Передача, хранение и обработка больших объемов научных данных : учебное пособие / А.А. Григорьев, Е.А. Исаев, П.А. Тарасов. — Москва : ИНФРА-М, 2024. — 207 с. — (Высшее образование). — DOI 10.12737/1073525. — ISBN 978-5-16-018850-8. — Текст : электронный. — URL: https://znanium.com/catalog/product/2051477 (дата обращения: 13.04.2024). – Режим доступа: по подписке.

17. Дадян, Э. Г. Методы, модели, средства хранения и обработки данных : учебник / Э.Г. Дадян, Ю.А. Зеленков. — Москва : Вузовский учебник : ИНФРА-М, 2022. — 168 с. — ISBN 978-5-9558-0490-3. — Текст : электронный. — URL: https://znanium.com/catalog/product/1834412 (дата обращения: 13.04.2024). – Режим доступа: по подписке.

ПРИЛОЖЕНИЕ А
Порядок создания проекта на основе Spark

Заготовка проекта с подключенной библиотекой Spark находится в файле scala-spark-project-template-master.zip.

После распаковки нужно импортировать проект в IntelliJ IDEA (рисунок 1), указав путь к папке с проектом (рисунок 2).

Рисунок 1 — Начальный экран IntelliJ IDEA

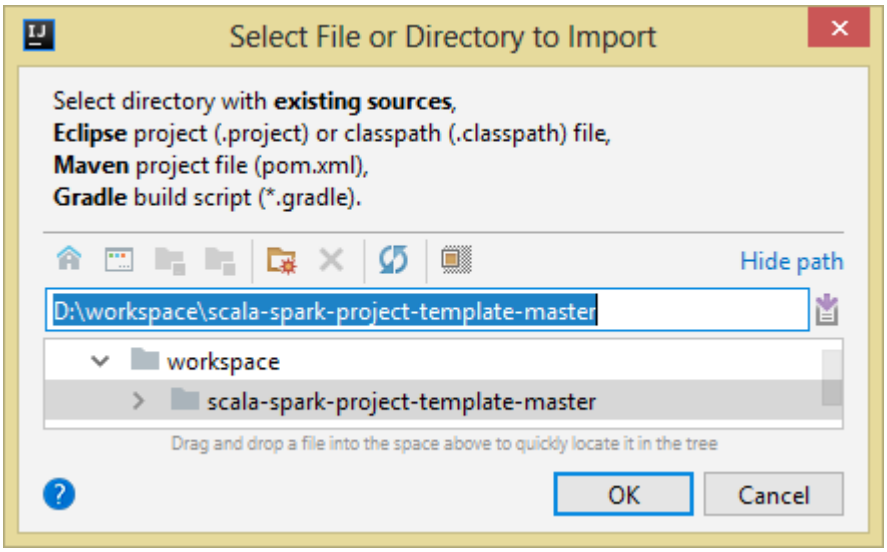


Рисунок 2 — Выбор папки с проектом для импорта

Далее нужно выбрать SBT в качестве сборщика проекта (рисунок 3) и версию JDK 1.8 (рисунок 4).

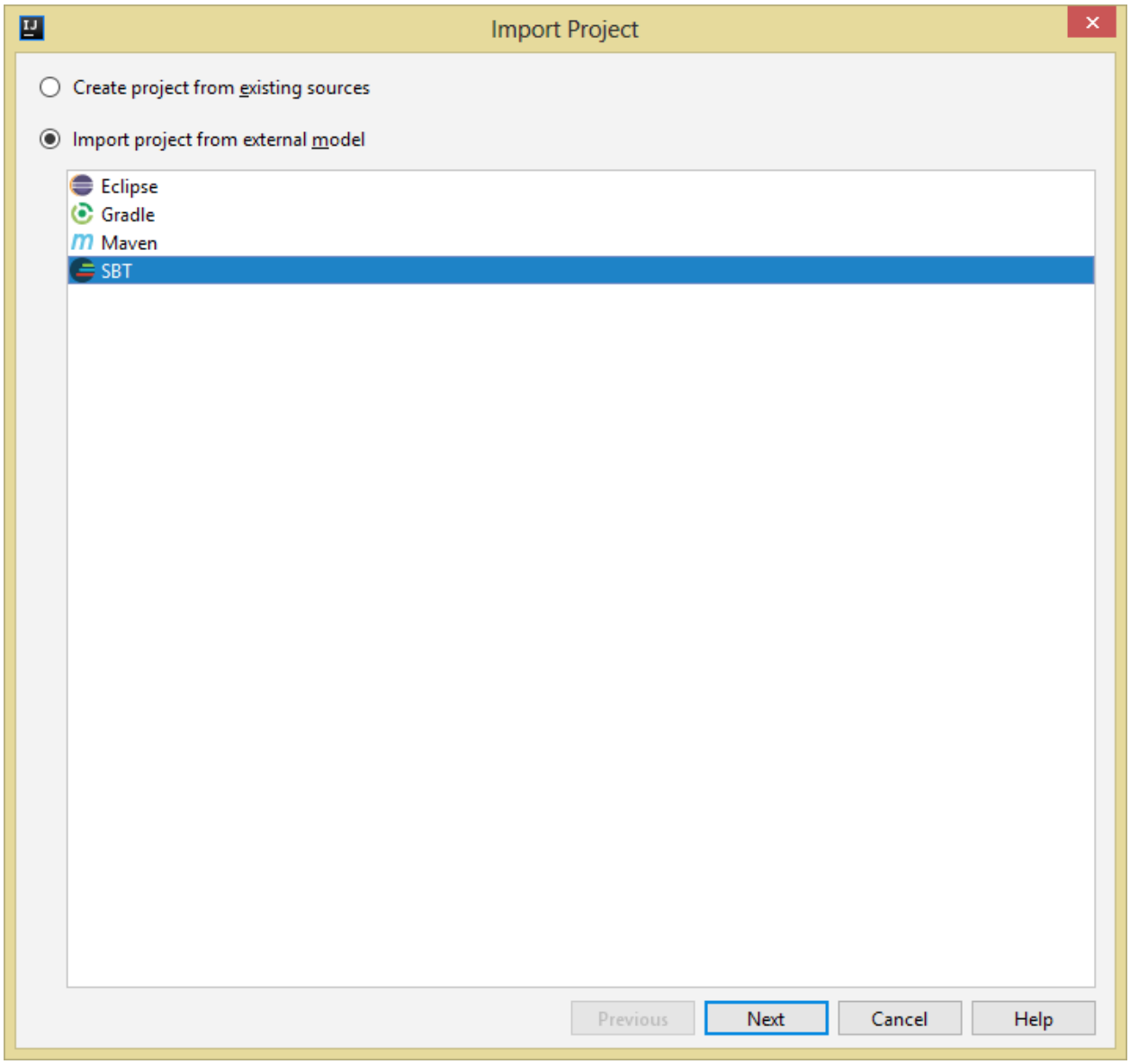


Рисунок 3 — Выбор сборщика проекта

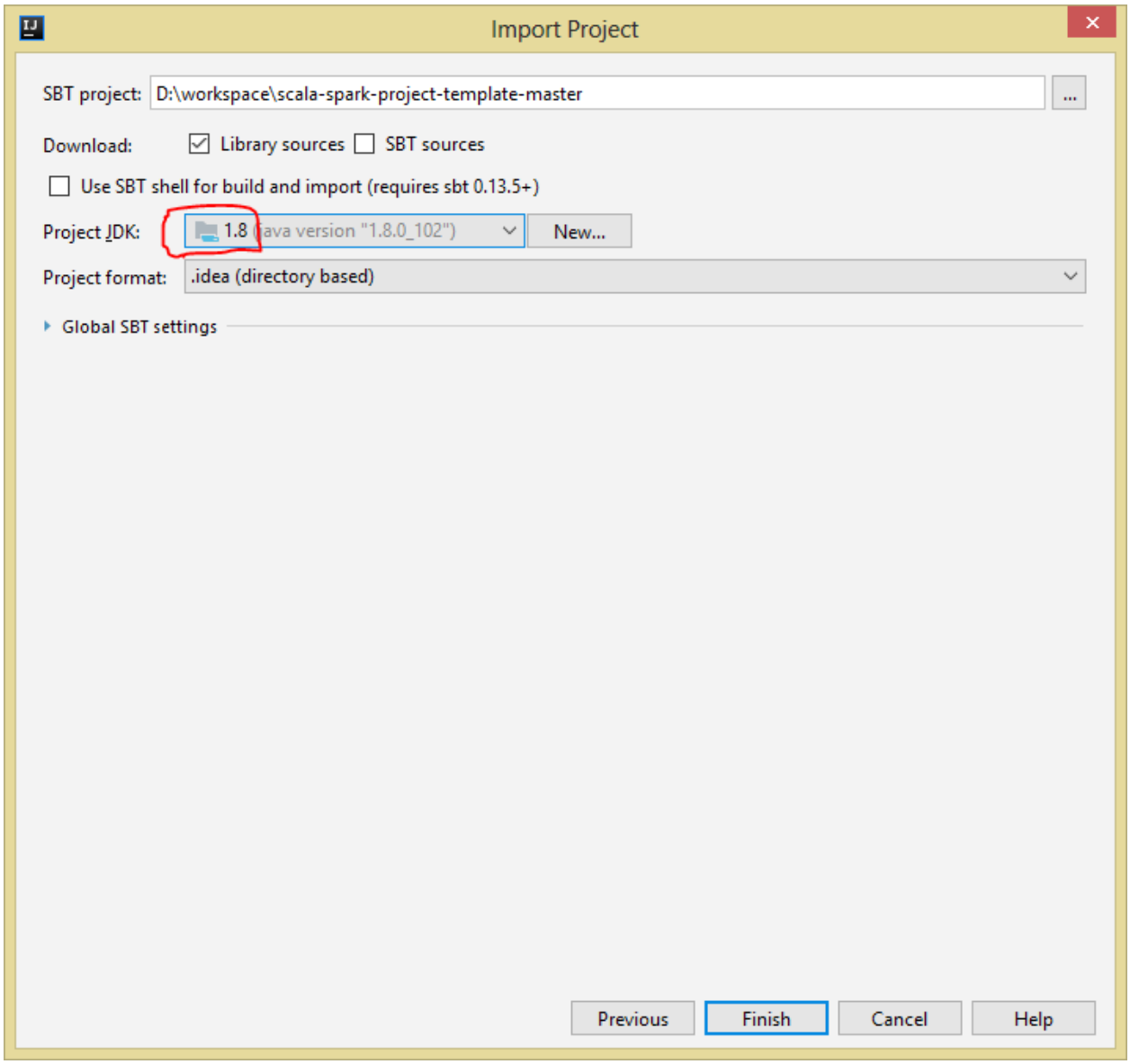


Рисунок 4 — Финальное окно импорта проекта

Затем начнется процесс скачивания необходимых зависимостей проекта (рисунок 5), который может занять некоторое время.

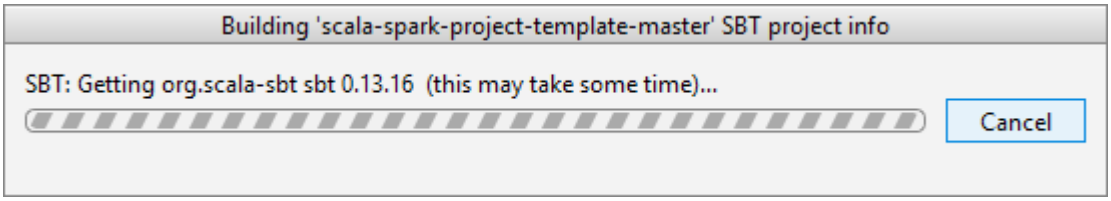


Рисунок 5 — Скачивание зависимостей проекта

После завершения скачивания зависимостей откроется основное окно IntelliJ IDEA и начнётся процесс индексации проекта. После завершения индексации можно запустить проект, нажав на зелёный треугольник рядом со строкой

«**object Application {**» или строкой «**def main(args: Array[String]): Unit = {**» (рисунок 6). В дальнейшем, после первого запуска проекта, можно делать запуск, нажимая на зеленый треугольник на верхней панели IntelliJ IDEA (рисунок 7).

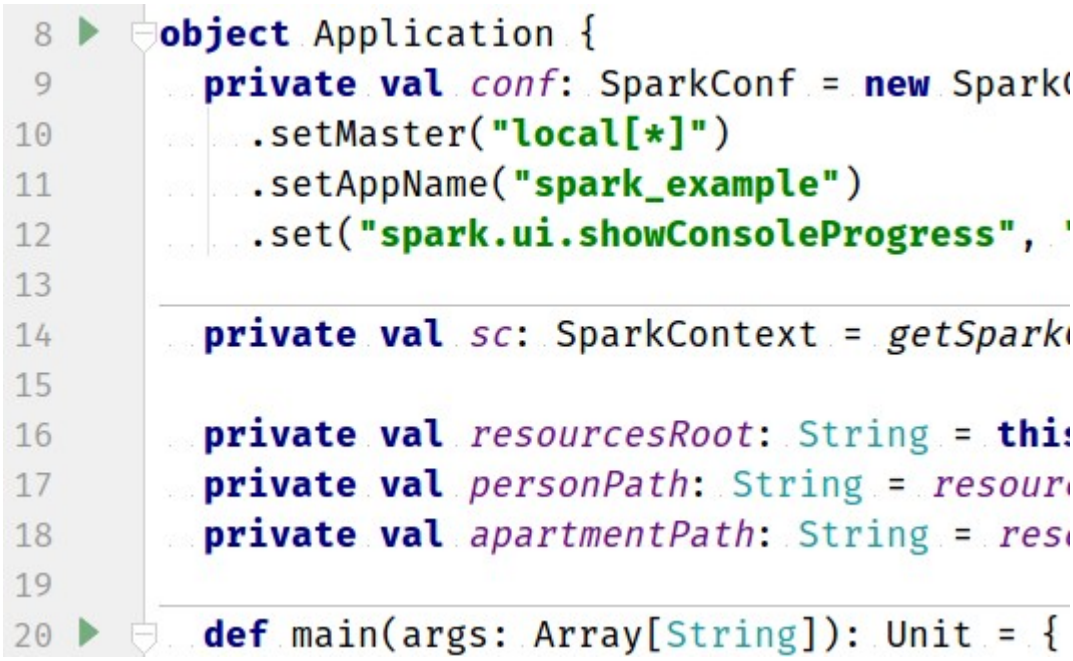


Рисунок 6 — Кнопки первоначального запуска проекта

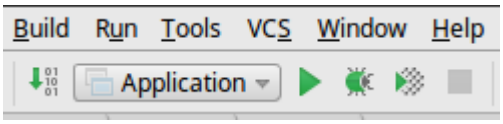


Рисунок 7 — Кнопка последующего запуска проекта